PID : Packaging - Integration - Development

An integrated software development process

Users Document

Robin Passama
CNRS Research Engineer
Robotic Department
LIRMM - UMR5506 - University Of Montpellier
passama@lirmm.fr

October 2013

Introduction

The main goal of this document is to provide a method that helps improving the overall quality of the code and applications produced by the robotics department of LIRMM. By improving the quality we aim at:

- simplifying the understanding and the modification of code produced by others.
- simplifying the reuse of software components we develop.
- pooling software developments between robotic teams.

To achieve this goal the Finding, Writing, Compiling, Testing, Version controlling and Documenting of the code are mandatory concerns. This document define in part 1, how the whole development process is organized, which tools are used and which concepts are handled. Then part 2 provides a detailed explanation of how PID is used and which processes it supports.

Contents

Part I
Fundamentals

The following subsections explain fundamentals that need to be understood to start working with the PID methodology. First of all, root concepts are:

- **package** : a package is the basic unit for code development, unit testing, source version controlling, deployment specification, documentation writing. A package provides some functional code that can be reused (libraries, executables, header files, scripts, etc.) and depends on other packages either for its compilation (static libraries, header files archives/folders) or for its runtime use (dynamic libraries, executables, script-like files). It can contains as well an entire huge software (e.g. operating system) as a very elementary piece of code (e.g. header files, a library, etc.).A package has two main different forms :
  - a source form. In this form, the package is a git repository that contains the package's source code. It is something "alive" that is continuously evolving along development process.
  - a binary form. In this form, the package is a collection of software artefacts (headers, configuration files, executables, libraries, documentation, etc.) deployed in a given place of a file system. It is something static (does not evolved) that is associated to a specific version of the package's source form. A binary package can be embedded into an archive in order to be easily retrieved and deployed on a file system.
- **workspace**: the workspace is the folder hierarchy in which the user develops source packages and deploys the binary packages he uses (third party or resulting from its own package deployment). The basic idea is to avoid to use system dependencies : every software artefact in the development process is then local and relative to the workspace, except of course artefacts bound to system dependencies.
- **package server** : a package server is a computer (accessible across the network) that hosts many packages (either source or binary). It centralizes the access to packages, and handles the rights of the users for each package. It is responsible of the global version control of packages. It can also provides tools to manage the development of packages it hosts (teams members, bugs and activities tracking/reports, wiki, version history and development branches, visualization, etc.).

The present document helps normalizing the development of packages inside workspace, and the way packages and workspaces are managed on development servers. As definition of concept and process is intrinsically bound to the concepts and process involved in the software tools used, the section first quickly present these tools. Then we define core concepts of the PID methodology based on those of the tools.

1 Tooling

For each development task in a package, a tool is used to achieve this task. To ease the portability of code, only **cross platforms** tools are used:

- **git** is used for the concurrent version control and patching of the code. It is also the tool used to deliver and retrieve the source code.
- **cmake** is used to manage the build process, but also deployment and test.
- **doxygen** is used to generate api documentation of source code.
- **latex** is the preferred language to write documents, since it allows to version the full content of file (raw text and structure of the document), as opposed to binary formats like Microsoft Word or Libre-Office that are not well handled by git.

Other tools used, like the compiler/linker (e.g. gcc) used with cmake, or the ssh agent used with git, or even development environment (e.g. Xcode, Eclipse) interfaced with cmake, git and doxygen, are supposed to be native (i.e. specific to the OS used) and so adapted by the user depending on the platform he uses and his own preferences.

2 Package

The package is the basic working unit for developers. A package :

- contains the functional source code.
- contains the tests source code.
- contains version control information files.

- contains the compilation files used to : build the source code and documentation, run tests, generate configuration files, install its resulting binary form in the workspace and generate an installable archive of its binary form.

The main idea is that a package is self-explanatory. It does not means it contains all code and artefacts it uses but it contains all information needed to satisfy its dependencies. In other words considering a given package, its installation process is done using this information and the information contained in its dependent packages.

One important concept when dealing with PID packages is the concept of **component** : it is a software artefact produced by a package. It may be either:

- a **library** : this is a component that can be reused by developers to build other components. We define three types of libraries : **shared** (runtime or load-time linked library), **static** (link time only linked library) and **header** (compile time only "linked" library).

- an **application** : this is a component for end-users or developers that define a runtime behaviour. We define three types of applications : application (made for end-user or developers when it is a runtime component), example (demonstrating how to use libraries) and test (used only internally to a package to test its other components).

2.1 Package Structure

A Package is generically structured according to the folder hierarchy defined below:

- the root folder of the package has the **name of the package**. This folder is basically a git repository which allows to manage concurrent work and version control on a package's content.

- the **.git** folder contains version control related information, automatically managed by the git tool.

- the root **.gitignore** file is used to exclude from version control some artefacts like temporary files.

- the **CMakeList.txt** file is used to describe how to build, install and test the whole package. It also contains meta-information on the package (authors and institutions, repository address, license, etc.).

- the **build** folder contains results from build process and contains two subdirectories: **release** and **debug**. Each of them contains the hierarchy of files and artefacts generated by the compilation process.

- the **src** folder contains sources files (.c/.cpp/.cc in C/C++) of libraries. Each subdirectory of **src** contains sources use to build one or more library and is itself hierarchically organized according to developers needs. Libraries provided by the package are defined by the CMakeList.txt file contained in the **src** folder.

- the **include** folder contains interface description files, typically exported (i.e. installed) headers files (.h, .hpp, .hh) in C/C++. Hierarchical organization in this directory is the same as in **src**. Non exported headers are let in the **src** folder, as they are not considered as a part of the interface of the package.

- the **apps** folder contains source files for applications, an application being an example of the usage of a library, a runtime component or a end-user software. Each subdirectory of **apps** contains sources for one or more built application and is hierarchically organized according to developers needs. Applications provided by the package are defined by the CMakeList.txt file contained in the **apps** folder.

- the **test** folder contains source files for test units. Each subdirectory of **test** contains sources for one or more test unit. Custom test programs and running tests applied to the package are defined by the CMakeList.txt file contained in the **test** folder.

- the **share** folder contains user written documents and some specific files used by the build process. Its contains different basic subdirectories :
  - the **doxygen** folder contains a "default" Doxyfile.in file that is used by doxygen to generate API documentation. This file can be modified by the user to add additional information to the generated documentation. The folder can also contain additional resources (like images), hierarchically organized according to developers needs, used by doxygen to integrate additionnal information in the API documentation.
  - the **cmake** folder contains cmake scripts (notably find scripts) that the package uses to find external resources like libraries. This is the place used only for very specific resources for which no default cmake script is available.
  - the **config** folder contains configurations files used by libraries and applications/tests of the package.
  - the **doc** folder contains "hand-written" documents (e.g. README files, tutorials, design documents, etc.).

  The **share** folder define a CMakeList.txt file that can be used to install resources of the **doc** and **config** folders.
- the **license.txt** file contains the license that applies to the source code produced in the package. This file is generated by the build process.

2.2 Package repository

Package repositories are GIT repositories, whose content is structured according to the previously defined pattern. GIT is used to version all text files used (C/C++ sources, cmake scripts, latex sources, etc.). Only source form of a package is a git repository not its binary forms.

2.2.1 Version Numbers as GIT Tags

A package is continuously evolving along time and git provide an internal version representation of this evolution. Nevertheless, this representation is so fine grained (each time a modification is committed) that it is not really understandable by persons not involved in the package development. That is why we need a version representation that can be either understandable by users and developers. These versions, called **release version** are defined according to a specific policy.

A **release version** can be viewed as a screen-shot of the git repository at a given time of package's life. It is associated to a number (called **release version number**) that is uniquely identifying the version. Technically, a version if represented as a **GIT tag** : a git tag memorizes a given state of the repository and is marked with a unique label that is used to retrieve this state. In our context the label of the git tag represents the release version number and the repository state pointed by the tag corresponds to the release version. The labelling of git tags representing release versions follows the pattern bellow:

- the release tags have the shape `vX.Y[.Z]`
- `X` is the major version number (starts with value 0). Change of major version number indicates that the code is no more completely backward compatible with previous versions. In other words, the interface of the package (some of the headers files it contains) has been modify in such a way that some function have change or disappeared, or the behaviour/meaning of existing functions completely changed. While X is 0 the version is considered as pre-released and so is not ready for use by third party developers.
- `Y` is the minor version number (starts with value 0). It indicates an improvement that is completely backward compatible with previous version with same major version number. In other words, the only a little change of existing behaviours occurred OR the interface of the package has been improved with new functionalities without breaking the way one use the older functionalities.

- `Z` is the patch version (starts with value 0). It represents a simple bug fix or a security fix. A patch changes nothing to the interface (no new behaviour/functionality) and modify only in a minimal way the internal behaviour.

Each time a new version of a package is released, its version number must be incremented according to the previously defined rules and a corresponding git tag is created. Here are some examples:

- 0.1.0 is the first pre-released version of the package.
- 0.Y.Z. are early development pre-released version of the package.
- 1.0.0 is the first release of source code.
- 1.2.0 is a release of source code backward compatible with version 1.0.0.
- 1.2.5 is a release of source code of version 1.2.0 with 5 bug/security fixes.
- 2.0.0 is a release that is no more backward compatible with 1.X.Y versions.

2.2.2 Development process with GIT branches

GIT branches are used to organize the development workflow by registering "increments" made in the repository. Increments are modifications of the repository content, either source code, documentation, etc. A git repository can have many branches representing parallel development work. Most of time developers create branches to isolate the work they do on a specific concern as regard of the software development. This work is keep isolated from the work made on other concerns until developers think this is the good time to integrate (or discard) them. From time to time GIT branches are created, deleted, and merged. Merging consists in registering modifications made in two or more branches into a single branch.

As GIT branches can be used to represent any type of concern their understanding can quickly become a real problem. That is why their usage is constrained according to a predefined pattern inspired from successful branching models. This pattern defines what is the meaning of branches, what they are use for and the way they are created and merged :

Main branches (see in figure 1) have infinite lifetime and must always be usable : their last commit must point to a state in which the package is compilable and executable with unit tests successful.

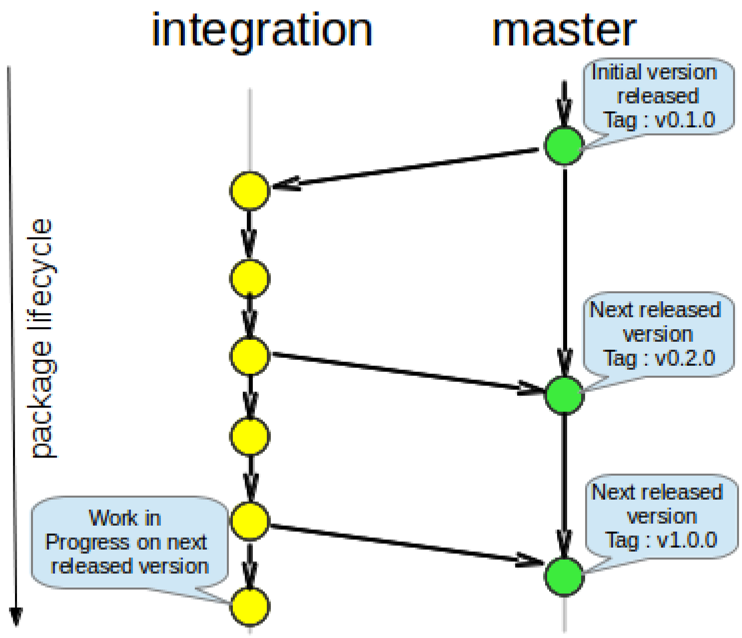

Figure 1: Permanent branches of a package

- The **master** branch contains the source code that always reflects a production-ready state. This branch is used to **tag** the released stable source code with version numbers but also to tag important intermediate states of the repository that reflect the development made for demonstrations and publications.

- The **integration** branch contains the detailed history of all the modification that have been realized on the repository. The source code of HEAD (pointer on the current state of the repository) always reflects a state with the latest delivered development changes for the next release. This is where any automatic nightly builds are built from, if any.

- When the source code in the **integration** branch reaches a stable point and is ready to be released, all of the changes should be merged back into **master** somehow and then tagged with an adequate release number.

Supporting branches are temporary branches, used to aid parallel development between team members, ease tracking of features and to assist in quickly fixing live production problems.

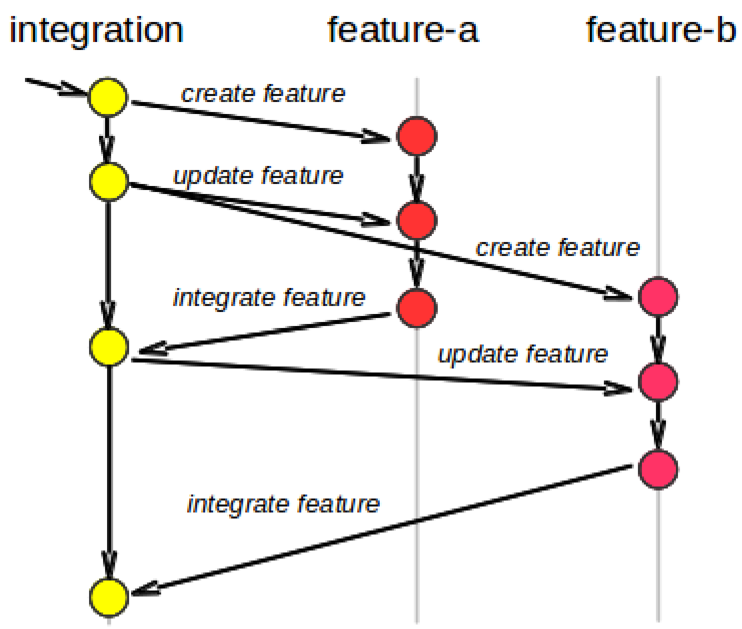

Figure 2: Relation between feature and integration branches

- **Features** branches (see figure 2) are used to develop new features /address new topics for the upcoming or a distant future release. Each feature branch must branch from the **integration** branch. A feature branch exists as long as the feature is in development, but will be merged back into develop (to definitely add the new feature to the upcoming release) or discarded (in case of a disappointing experiment). During its lifetime a feature should be updated from time to time with the modifications contained in the **integration** branch (issued for instance from the integration of other features). Doing so, the final merge of the feature will be more easy as the state of the feature will be not to far, in terms of importance of modifications, from the state of the **integration** branch.

- **Hotfixes** branches (see figure 3) arise from the necessity to act immediately upon an undesired state of a released version. When a critical bug in a realease version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the **master** branch that marks the production version. Hotfix branches are used to allow team members to continue their work (on the **integration** and **feature** branches), while another person is preparing a quick bug/security

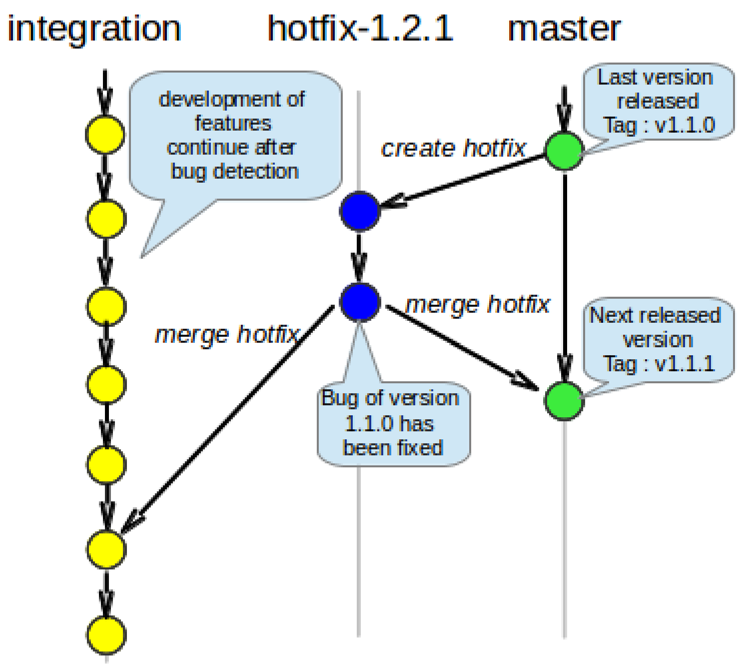

Figure 3: Relation between hotfix and integration/master branches

fix. When bugs are solved, the bug fix must be merged back into both **master** and **integration** branches. When merged the **master** branch is tagged with a new *patch version number*.

Naming Conventions:

- a **feature** branch name starts with "feature-" and ends with the name of the feature (given by developer). example : `feature-newfunctionnality`
- a **hotfix** branch name starts with "hotfix-" and ends with the new *patch version* of the released version. example : `hotfix-1.2.1`

2.2.3 Collaborative working with GIT repositories

Now that the way the package is structured and is evolving the last part of this section consist in defining how people involved in the package's life cycle work together. The first concern is to define the **privileges** owned by these persons as regard of the repository usage. To do so we first define the three roles associated to each package:

- **users** are people using the package but not involved in its development.
- **developers** are users involved in the development of a package.
- **administrators** are developers with additional privileges, as they are considered as responsible of the package by users of the package.

Privileges associated to each role are managed inside the **package server** that hosts the package:

- each package repository is associated to three *groups*, each one representing a role previously defined. For example, for a package "a-pack" , there are groups "a-pack-users", "a-pack-developers" and "a-pack-administrators". Each group provides specific privileges on the package repository.
- users registered inside the **package server** may be affected to one of these *groups*. Doing so, these users obtain corresponding privileges on the package repository.
- the repository can be set "public" so that anyone is considered as a user. In this case the users group may be not useful and can be let undefined.

The basic scheme for collaborative working between package developers is presented in figure 4. Given a package, this package has an **official GIT repository** that is deployed on a given **package server**. This repository is *official* because it centralizes information of the package and is public, which means that all concerned people (team, laboratory or more generally anyone registered in the server) can access to it. The access in itself is restricted with respects to roles:

- registered **administrators** have read/write access on the official package.
- all other registered **users** have read access.
- unregistered people may have read access (i.e. open source repository).

During development, one or more **private repositories** of the package can be created by *forking* the official repository. Depending on the official repository either only registered administrators (private package), developers (protected package), users (public package), or anyone (open package) can *fork* the official repository. Then the creator in addition of the administrators

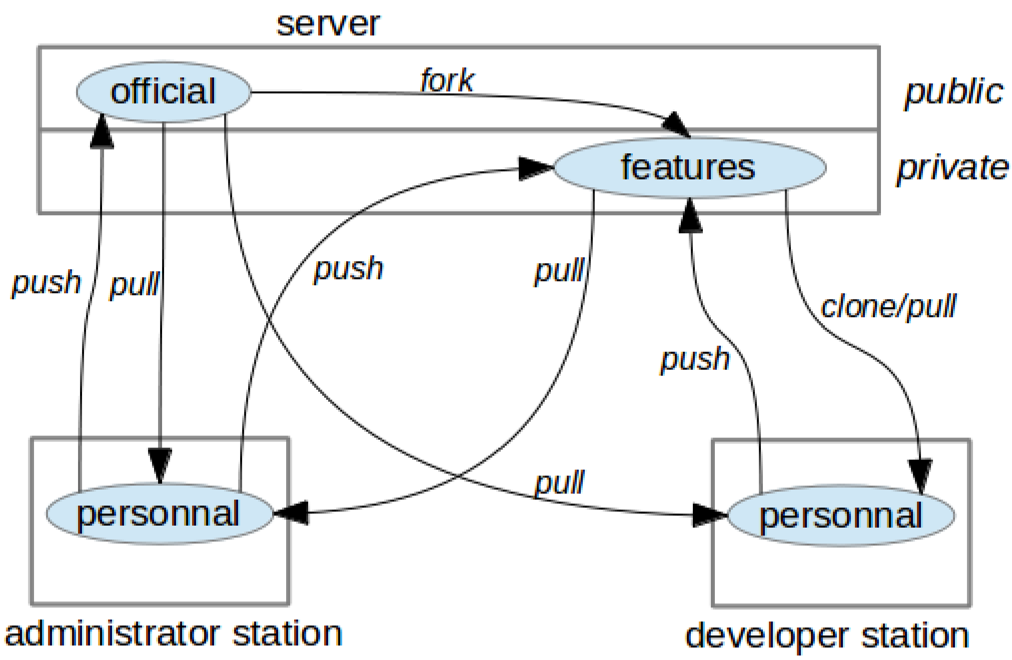

Figure 4: Collaboration between developers and repository

of the official, become **administrators** of the private repository. Then they can register new **users** and **developers** for this repository, with following privileges:

- registered **administrators** and **developers** have read/write access.
- registered **users** have read access.
- unregistered people have no access.

These repositories help structuring the development process by grouping developers work around a common repository while not immediately impacting the official one. This lets the time to the **administrators** to check if everything is OK and that modifications can be merged back into the official repository. The main purpose when using **private repositories**, is to separate the work on the same package made by different groups. For instance, a PhD student working on its demos on one side and a team with external people working in a common research project on the other side.

Administrators of the official package are the only ones that can update the **official repository** with modifications made on private repositories. That is why they are responsible of *version releasing* (see previous section):

they check the work done by developers and decides if the code is stable enough to release it. Furthermore, if there are many private repositories of the same package they act as brokers of changes made in separate pools of developers.

3 Workspace

The workspace is the place where developers work with packages (either those they develop or those they simply use). Technically, workspaces are local folder hierarchies in any station (server/demo/developers computers) that works with PID packages. A workspace is the place where packages repositories are created, developed and installed, and referenced given a pre-determined pattern. As compared to classical "do as you wish" installation, the "workspace way of doing" as several advantages:

- there is a reference path in the deploy process in such a way that classifying and finding software artefacts is made easy.
- there is no need to use system dependencies intensively for developed packages. As far as possible, every dependency should be satisfied by workspace content, except for system dependencies that cannot be packaged into workspace.
- developers can handle many version of a package in the same build/execution environment, without risking to break OS dependencies. Furthermore they can used multiple versions of the same package in the same time.

3.1 Workspace organization

3.1.1 Overview

A workspace is a folder (preferably named "workspace") with the following structure:

- the **.git** folder contains version control related information, managed by the git tool.
- the **install** folder contains packages' binaries installed by the user.
- the **packages** folder contains all the packages developed by an author. Each sub-folder of **packages** is a local repository of a given package as presented previously.
- the **external** folder contains all the external packages used by the installed package (source or binary). Each subdirectory corresponds to an external package, whose content is in turn completely undefined.
- the **.gitignore** file is used to exclude **install**, **packages** and **external** folders content from version control. This is mandatory since these folders content is supposed to be **purely local a a user workstation**.

- the **share** folder contains important files used to manage packages. The **docs** sub-folder contains documentation including the present document ; the **cmake** folder contains all scripts and data required to describe, build, install, deploy and configure packages: the **find** sub-folder contains cmake find script for commonly used external packages ; the **system** sub-folder contains generic scripts ; **patterns** contains cmake pattern files used ; **references** contains cmake script files with meta-information about available packages ; **licenses** contains cmake script files containing available license description;

3.1.2 Installing binary packages in the workspace

The **install** folder contains installed binary version(s) of any package used by a user. Each of its direct sub directories is a folder representing a given installed package, that itself contains :

- as many folders as there are concurrent binary versions of the package installed. The name of the version folder reflects the installed package version it contains (e.g. : 1.3.5 ; 1.2.0, etc.). Version folder can be also local (correspond to an install from a source package currently developed in the workspace), in such a case their name is of the form : own-0.2.5, own-1.0.0, etc.)

- an **installers** folder that contains all installable binary archives with each archive that corresponds to a specific version, for a specific system (linux, mac), in a specific mode (release, debug).

Each version folder is organized according to the following structure:

- the **bin** folder contains all executables provided by the package, except tests, in both debug and release modes. In other words it contains the result of the compilation of its corresponding package repository **apps** folder.

- the **include** folder contains the exported interfaces of libraries. Basically, its direct sub-folders are libraries' root folder hierarchically organized the same way as in the package repository **include** folder.

- the **lib** folder contains libraries provided by the of the package in both debug and release modes. In other words it contains the result of the compilation of its corresponding package repository **src** folder.

- the **share** folder contains documents and scripts that are useful to the package users: the Use<Package><Version>.cmake file is a specific script file used to identify elements and dependencies of the binary package ; its **doc** sub-folder contains API documentation generated by **doxygen** ; the **cmake** sub-folder contains cmake scripts files that are required to use the package, like find scripts used to configure external packages ; the **config** sub-folder contains installed element contained in the corresponding config folder of the package repository **src** folder ; the **doc** sub-folder contains "hand-written" documents installed with the package.

- the **license.txt** file describes the license that applies to the software. This is a copy of the license file in package repository.

3.2 Workspace repository

The **official workspace** is a git repository that can be modified only by **administrators** of the server. It contains cmake scripts used notably to:

- reference available packages (repositories and binary archives). Each time a new package is created it is referenced into the local workspace (by an administrator) and then changes in the official workspace are committed so that anyone can know the existence of this new package and can retrieve its hosting server. This does not mean that this person can use binary version or repository of this package, since this is bound to his access rights to the servers (at least one can know which person to call to get the rights).

- provide available licenses description. These descriptions will be used in packages description.

The **official workspace** can be forked into **private workspaces**, exactly the same way as an **official package** (see figure **??**), to provide a common workspace for a team developing one or more packages at the same time. Once created new **users** and **developers** can be added to the private workspace repository. Private workspaces will be updated (by **developers** and **administrators**) with new references to packages, new licenses and new find scripts, while new packages are implemented. Then **official workspace** administrators can update its content, at any time, with the modifications made inside the private repository.

Part II
Usage

This part of the document provides a detailed description on the usage of PID methodology's core concepts and tools. The first section describe how to use git tool to manage the collaborative work-flow. Second section describes how to use CMake tool to describe packages. Finally last section provides guidelines and advises to put in place good development practices.

4 Work-flow management with git

The purpose of this section is to define how to use the git tool to manage the whole life cycle of a project managed according to the principles of the PID methodology.

4.1 Installing Workspace and Packages

The first phase when starting development consists in installing a workspace on the local station of a developer or administrator, and configuring it adequately. Figure 5 provides a general overview of this operation.

4.1.1 Getting a workspace

The initial phase for any user is to **get a workspace on his local station**. More precisely a user needs to get a workspace repository that is connected to a **private workspace** (see section 3.2). This **private workspace** is used by a group of developers that work on same packages, for instance members of a same research team, members of a same research project, or even a single person for a PhD student whose work is isolated from the others. Only workspace administrators can create new private workspace repositories since their job is to know which persons can / have to work together.

This initial phase simply consists in either (see figure 5):

- if a **new group** is created for the developer:
    1. an administrator *forks* the **official workspace repository** to get a **private workspace** on a server.
    2. the developer *clones* the **private workspace** in his local file system.

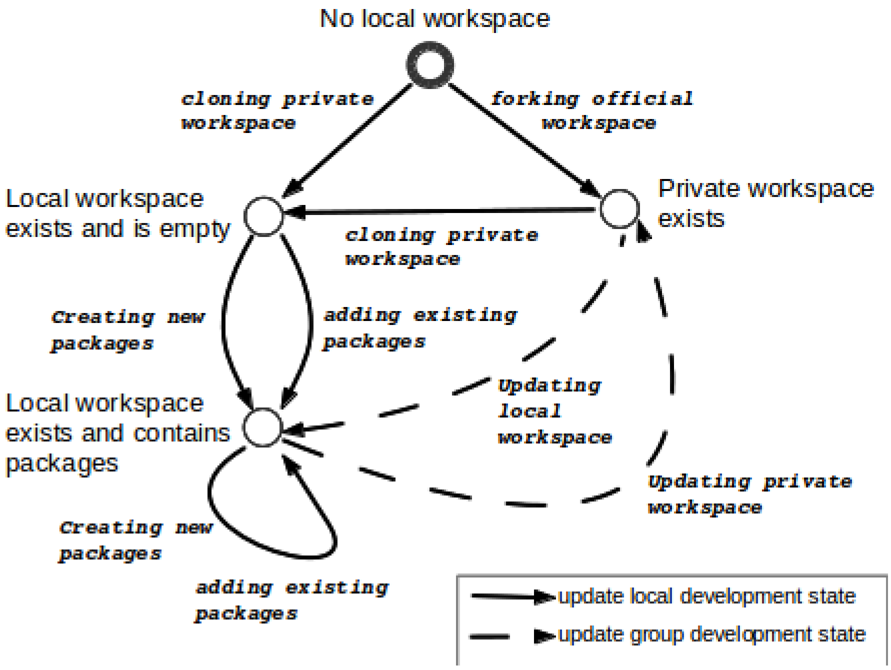

Figure 5: State diagram showing how to deal with workspace

- if the developer will join an **existing group**:
  1. an administrator gives to him the privileges to access the **private workspace** of the group.
  2. the developer *clones* the **private workspace** in his local file system.

Cloning and forking are similar operations, based on the clone action of the git tool like:

```
git clone <account>@<official server>:workspace.git
```

The main difference between both actions is where they take place. *Cloning* is supposed to be done on local station of developers while *Forking* is a clone action done in the official server. *Forking* operation requires more actions like connecting to server and managing rights for developers that can access to the private repository: this can be easily achieved with Web based tool like GitHub or GitLab, but this is beyond the scope of this document.

When the local workspace repository has been cloned, the developer's workspace is empty, which means it only contains available licenses description in **licenses** folder, references to available packages in **references** folder but its **packages** and **install** folders are empty (see section 3.1.1).

4.1.2 Adding packages to the workspace

Since a **developer** is supposed to work on at least one package (but often more than one), he must add packages to its local workspace. To do so, he can create a new package and/or use existing packages.

Creating a new package:

1. the developer creates a new **private package repository** on a **package server**. For the moment no **official package repository** exists for that package.

2. On his local station he then *clones* the **private package repository** and initializes it the following way:

   ```
   cd <workspace path>/packages
   git clone <account>@<private server>:<new package>.git
   cd <new package>
   cp -R ../../share/patterns/package .
   git commit -a -m "initialization of package done"
   git tag -a v0.0.0 -m "creation of package"
   git push origin master
   git push origin tags/v0.0.0
   git checkout -b integration master
   git push origin integration
   ```

3. then the developer starts working with its local and the private repositories (TODO add ref), he is the only person that can access it. At the beginning the package has a generic structure as in section **??** but is empty: the first thing to do is to define package using CMakeList.txt files (see section 5) and to add content (i.e. source code).

4. when the developer has reach a state in which the package can be delivered to the group he then:

   (a) generates/installs the **reference file** and the **find file** for that package using adequate CMake configuration (see section 5.5): the **reference file** and the **find file** are respectively put is put into **references** and **find** folders of its local workspace (see section 3.1.1).

   (b) updates the **private workspace repository** of its group by doing:

```
cd <workspace path>
git add --all
git commit -m "adding new package : <package added>"
git push origin master
```

Using existing packages:

1. the **administrator** gives to the **developer** the adequate privileges on the used **private package repositories** (those developed by its group, or only part of them).

2. On his local station the **developer** then *clones* the **private package repository**:

   ```
   cd <workspace path>/packages
   git clone <account>@<private server>:<used package>.git
   ```

3. then the developer starts working with its local and the private repositories (TODO add ref).

Important remark: all packages developed by a group must be **referenced** with the references files available in the **private workspace repository** of this group. Each package developed is itself connected to a given **private package repository** as explained in section 2.2.3. Doing so each group work on an isolated development environment. Environments of groups are synchronized in two ways:

- **private package repositories** are updated with modifications made on **official package repositories** and reversely. These updates are completely managed by administrators who have the rights to modify **official package repositories**. This synchronization can be done progressively during packages development process.

- **official workspace repository** is updated with modifications made on **private workspace repositories**. These updates are completely managed by administrators who have the rights to modify **official workspace repository**. This synchronization is mainly done when development process in a group has reached a stable state and after the first release of packages that have been created by a group.

4.1.3 Installing other required packages in the workspace

Whether a developer creates or use an existing package, this later can have dependencies with other packages and with system or external packages (see section 5.1.2). System dependencies are installed using standard install process provided by the OS (e.g. apt-get on Ubuntu/debian distributions) while external dependencies are installed "by hand". Package dependencies are installed using PID specific CMake procedure based on the **reference files**.

Each **reference file** of the workspace provide meta-information on a given package, notably the address of its git repository and addresses where to find binary package versions relocatable archives of this package (see section 5.7). These informations are used to install adequate version of a required package at let two alternatives to install a package dependency:

- the git repository is accessible (at least in reading) by the developer and he can then clone and build it.

- the adequate binary archives can be downloaded by the developer and he can then directly install them.

In both cases, the process can be automatically managed by PID specific CMake scripts, which is the far most simple way. Otherwise the user can do this "by-hand", but this is beyond the scope of this document (i.e. reserved to expert users).

The only important thing to understand is the role of **reference files** and how they must be managed. In **private and local workspace repositories**, there is a set of **reference files that simply represent all available packages**. These reference files contain information on git repositories of packages, structured according to the following pattern:

- a reference file of a package that is not developed by the group contains the address of the **official package repository**. Members of the group don**'t have write access** to this repository and **may have read access or not**. It also contains addresses of downloadable binary package version relocatable archives whose access is generally open to all members of the robotic department or even sometime beyond (opened to partners, public access).

- a reference file of a package that is developed by the group contains the address of the **private package repository**. Each time a member of the group *forks* an **official package repository** into a **private package repository** he has to:

    1. *clone* the **private package repository** into his local workspace.

2. change the address of the package repository (official package repository address) with the address of the newly created **private package repository** (see section 5.1.1 and section 7.4).
3. generate the new reference file using PID specific CMake package configuration (see section 5.5).
4. update the **private package repository**:

```
cd <workspace path>/packages/<new private package>
git checkout integration
git add CMakeList.txt
git commit -m "updating git address of the repository"
git push origin integration
```

5. update the **private workspace repository** (whose references folder has been modified with an updated reference file for the package):

```
cd <workspace path>
git add share/cmake/references/Ref<package>.cmake
git commit -m "changing git repository address for <package>"
git push origin master
```

When done, installation of packages repositories for other members of the group will be quite simple and will target adequate **private package repository** (for whose members will have right to access in read/write). Only administrator will be responsible to avoid doing mistakes when merging **private package repositories** into **official package repositories** and **private workspace repositories** into **official workspace repository**. Their main task will be (1) to remove references to **private package repositories** in package CMakeList.txt files and (2) to merge adequately **reference files** from private and official repositories.

4.2 Collaborative package development

All **developers** only work around **private repositories** of packages, while **administrators** are responsible of the update of **official repositories** according to the changes done in private repositories. Most of developments in **private repositories** are made in **feature branches** and merged into the **integration branch** (see section 2.2.2).

4.2.1 Handling feature branches

- Creating a feature branch (derived from *integration* branch): beginning of the day):

  ```
  git checkout -b <feature name> integration
  ```

- When starting modification of a feature (for instance at the beginning of the day):

  ```
  git checkout feature-<feature name>
  // now working on <feature name> branch
  git pull origin feature-<feature name>
  // now <feature name> branch is up to date
  ```

- During development of a feature, developers need to frequently "save" their work locally:

  ```
  git add <modified or new files>
  git commit -m "<telling modifications>"
  ```

- When finishing modification of a feature (for instance at the end of the day or when important modifications have been finished and committed):

  ```
  git pull origin feature-<feature name>
  git pull origin integration
  // update with evolutions of the integration branch
  git push origin feature-<feature name>
  ```

During development on a feature branch, a developer may need to test some ideas without disrupting the work of all people working on the feature. In that case, he can work with a **new local branch** forked from feature branch. Modifications in this branch stay always local to the developer station. Furthermore modification should not be too big, these branches are used to test some ideas or to do bug fixes, not for developing big features (this is the role of feature branches) !

- Creating a local branch for testing an idea:

```
git checkout feature-<feature name>
git pull origin feature-<feature name>
//=> solving potential conflicts and testing
git push origin feature-<feature name>
git checkout -b <idea> feature-<feature name>
```

- During development of the idea, frequently save the work locally:

```
git add <modified or new files>
git commit -m "<telling modifications>"
```

- When idea is OK and can be integrated to the feature:

```
git pull origin/feature-<feature name>
//=> solving potential conflicts and testing
git rebase origin/feature-<feature name>
git checkout feature-<feature name>
git merge <idea>
git branch -d <idea>
git push origin feature-<feature name>
```

- otherwise if the idea is not good:

```
git checkout feature-<feature name>
git branch -d <idea>
```

4.2.2 Integration of features

During development process, features are integrated as soon as their development is finished and their tests are OK. Integration simply consists in merging feature branch into integration branch:

1. Merging the feature in the development branch (only developer responsible of the merge):

```
git checkout feature-<feature name>
git pull origin feature-<feature name>
//=> solving conflicts and testing, now feature is OK
git pull origin integration
//=> solving conflicts and testing, now feature ready to be merged
git checkout integration
git merge --no-ff feature-<feature name>
//=> solving conflicts and testing, now integration OK
```

2. Deleting the feature branch (only developer responsible of the merge):

```
git push origin integration
git push origin --delete feature-<feature name>
```

3. Updating local repositories after merge in the private repository (all developers of the repository) :

```
git remote prune origin
git branch -d feature-<feature name>
```

Remarks:

- The development of features is made in parallel and they are merged indirectly in the **integration** branch at the very end, one at a time: features don't synchronize until merge. This let the possibility to developers to change some parts of the API/structure without immediately impacting the work made on others features.
- The best way is to first create an initial feature branch that puts in place the general basis of the package (API, basic class hierarchy, etc.). Then, when this feature has been merged in **integration** branch, parallel development into many feature branches can start.
- When merging, the resolution of conflicts **must be realized in feature branch** to avoid any problem in the integration branch while conflicts resolution takes place.

4.2.3 Releasing a Package Version

Releasing package versions is the responsibility of administrators. It generally consists in merging **integration** branch into **master** branch and then tagging the result in master with a version number (see section 2.2.1).

Version tags handling is made the following way:

- creating a version number (annotated tags):

```
git tag -a v1.2.3 -m"<small description of the version>"
git push origin tags/v1.2.3
```

- listing all available versions:

```
git tag -l 'v*'
```

- showing the information of a given version:

```
git show v<version number>
```

- getting the released version in package history:

```
git checkout tags/v<version number>
```

It sis also possible to use tags during on **integration** branch, for instance to memorize code state matching demos or papers. Then use `p<journal or conf>-<date>-<author>` pattern tags for papers and `d<name of demo>-<date>-<author>` pattern tags for demos.

Administrator's repositories are most of time more complex than simple developers ones as they reference the **official repository** and at least one **private repository**. Administrators must so manage multiple references to repositories:

- Creation of the local repository of a given package:

```
cd <path to workspace>/packages/
git clone <package official repository>
```

- Registering a **private repository** for a given package:

```
cd <path to workspace>/packages/<package>
git remote add <private repository name> <repository address>
```

- Getting the state of a private repository, without merging the result with administrator's local branches:

```
git fetch <private repository name>
```

```
git remote rm <private repository name>
```

The release process takes place into the **administrator**'s stations.

1. Update administrator's station with modifications contained in integration branch of private (named `private`) and official (named `origin` for administrators) repositories :

```
git checkout integration
git pull private integration
//=> solving conflicts and testing
git pull origin integration
//=> solving conflicts and testing, now ready to be merged
```

2. Merging master and integration branches in local repository of administrator's station:

```
git checkout master
git pull private master
git pull origin master
//=> solving conflicts and testing
git merge --no-ff integration
//=> solving conflicts and testing, ready to be released
```

3. Updating version number. Setting the adequate version number in the package's CMakeList.txt to get `<released version>`, rebuilding package and then:

```
git commit -a -m "Bumped version to <released version>"
git tag -a v<released version> -m"<description of version>"
```

4. Updating official and private repositories:

```
git push origin integration
git push private integration
git push origin master
git push private master
git push origin tags/v<released version>
git push private tags/v<released version>
//=> updated branches and tags of private and official repositories
```

4.2.4 Developing a hotfix

Creating a hot fix is always made on demand of an administrator, but can be realized either by himself or a developer. The process is quite the same as for features:

1. Creating the hotfix branch in the private repository, derived from version <old version>:

```
git checkout tags/v<old version>
git checkout -b hotfix-<new patch version>
```

2. During the bug correction, committing locally:

```
git add <modified files>
git commit -m "<commit message>"
```

3. Saving work in the private repository:

```
git push origin hotfix-<new patch version>
```

4. Updating version number, when bug or security problems have been solved. Setting the adequate version number in the package's CMakeList.txt to get `<new patch version>`, rebuilding package and then:

```
git commit -a -m "Bumped version to <new patch version>"
git push origin hotfix-<new patch version>
```

5. Merging hotfix branch with master:

```
git checkout master
git merge --no-ff hotfix-<new patch version>
git tag -a v<new patch version> -m"<description of version>"
git push origin master
git push origin tags/v<new patch version>
```

6. Merging the bug correction with the integration branch:

```
git checkout integration
git merge --no-ff hotfix-<new patch version>
//=> solving conflicts and testing
git push origin integration
```

7. Deleting hotfix branch on local and private repositories:

```
git branch -d hotfix-<new patch version>
git push origin --delete hotfix-<new patch version>
```

8. Releasing the patch version in the official repository. This is the role of administrators that do that on their own station:

```
git fetch private
git checkout integration
git pull private/integration
//=> solving conflicts and testing
git push origin integration

git checkout master
git pull private/master
git push origin master
git push origin tags/v<new patch version>
```

5 Package development with CMake

When developing a package one need to handle information such as:

- *meta-information*: who is involved in its development ? what is its general purpose ? where to find this package on the network ? what is the license of the code ? etc.

- *build-related information*: what are the components provided by the package and how to compile/link them from source code ? How components are tested ? what is the version of the package ? what are its functional dependencies ?

- *functional information*: this is the source code of the components and other associated files like configuration files if any required.

The whole package development is organized and described with cmake, and the CMakeList.txt files used contain the whole *meta-information* and *build-related information* used by the package. Each package contains several CMakeList.txt files:

- the root CMakeList.txt file (direct leaf of the package source repository, see section 2.1) is used to define *meta-information* and *dependencies* of the package.

- the CMakeList.txt file contained in the **src** folder defines the **library components** (see section 2).

- the CMakeList.txt file contained in the **apps** folder defines the **application components** (see section 2).

- the CMakeList.txt file contained in the **test** folder defines the **test components** (see section 2) and running tests (using these components or others).

- the CMakeList.txt file contained in the **share** folder defines additional files to install, like configuration files used by libraries and/or applications (if any), documents, etc.

Each of these CMakeList.txt files must follow a predefined pattern. This pattern is mainly influenced by the use of PID specific CMake functions. The following subsections present examples on how to use these functions together with more classical CMake code in order to completely define a PID package.

5.1 Package's root CMakeList.txt file

5.1.1 General Meta-Information

Let's suppose we define a package with the name "the-testpack-b", its root CMakeList.txt file could look like this.

```
PROJECT(the-testpack-b)
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.11)
set(WORKSPACE_DIR ${CMAKE_SOURCE_DIR}/../.. CACHE PATH "root of
the packages workspace directory")
list(APPEND CMAKE_MODULE_PATH ${WORKSPACE_DIR}/share/cmake/system)
# using generic scripts/modules of the workspace
include(Package_Definition)

declare_PID_Package(
    AUTHOR      Robin Passama
    INSTITUTION LIRMM
    YEAR        2013
    LICENSE     CeCILL
    ADDRESS     git@idh.lirmm.fr:perso/passama/the-testpack-b.git
    DESCRIPTION  test package B for PID
)

set_PID_Package_Version(1 1)

# adding some binary packages references
add_PID_Package_Reference(
    BINARY VERSION 0 1 0 SYSTEM linux
    URL
    http://lirmm.lirmm.fr/FileX/get?auto=1&k=rfYTf1gkpI5XtEpQWVA
    http://lirmm.lirmm.fr/FileX/get?auto=1&k=oMyg4JVeeKYWpqwEFxE
)

add_PID_Package_Reference(
    BINARY VERSION 1 1 SYSTEM linux
    URL
    http://lirmm.lirmm.fr/FileX/get?auto=1&k=DYt2j35Kw8ozOgHfoVA
    http://lirmm.lirmm.fr/FileX/get?auto=1&k=zEx02N4KWfzDWPTxiO
)
```

Exactly as in any CMake project, the CMake package is define with the PROJECT keyword. The project's name **is the name of the package**

root folder (and so the name of the repository). The remaining lines until the `declare_PID_Package` macro call must be let **unchanged** (initialization of the PID specific cmake scripting system).

Then comes the `declare_PID_Package` macro, which is mandatory. This macro defines general meta-information on the package, and should not change a lot during package life cycle:

- the main author (AUTHOR keyword) and its institution (INSTITUTION optional keyword), considered as the maintainer of the package. The main author is an **administrator** of the package (see section 2.2.3).
- the YEAR field helps defining the package's life cycle range. For instance one can input a field such as "2009-2013".
- the LICENSE field is used to specify the license that applies to the code. This license must be defined in the workspace (see section 3.1.1) in the form of a **cmake script license file**.
- the ADDRESS field is used to specify the address of the **official GIT repository** of the package (see section 2.2.3).
- the field DESCRIPTION must be filled with a short description of the package usage/utility.

Then the user fill other meta-informations that evolve during project life cycle:

- the `set_PID_Package_Version` function is used to set the currently developed version of the package. It take at least a MAJOR and MINOR numbers and optionally a PATCH number as arguments (default value for PATCH number is 0). The version number thus follows the same pattern as git **release versions**. Before a version is released with a git tag (see section 2.2.1) this version number must be set adequately so that git tag matches cmake package version. Generally, the best way to do is set the version number used in the CMakeList.txt with the number of the next version to release.
- the `add_PID_Package_Reference` function is used to register a downloadable binary version of the package. The VERSION keyword specify the version with MAJOR MINOR PATCH numbers. The SYSTEM

keyword specifies the target operating system for which the binaries have been built. The two addresses after the URL keyword specify where the binary package version can be downloaded either in release (first address) and debug (second address) modes.

5.1.2 Dependencies with other packages

The last part of the root CMakeList.txt is used to manage dependencies between the current package and other packages it depends on. It could look like this:

```
# finding used packages
find_package (Boost REQUIRED)
find_package (the-testpack-a 1.0 REQUIRED lib-b-sh lib-x)

# declare a dependency over Boost (by default boost root is the
# /usr system install dir)
if(Boost_DIR-NOTFOUND)
     set(BOOST_ROOT /usr)
endif()
declare_PID_Package_Dependency(PACKAGE Boost EXTERNAL
${BOOST_ROOT} VERSION "${Boost_MAJOR_VERSION}.
${Boost_MINOR_VERSION}.${Boost_SUBMINOR_VERSION}")

#declare a dependency over the-testpack-a PID package
declare_PID_Package_Dependency (
                PACKAGE the-testpack-a
                PID VERSION 1 0
                COMPONENTS lib-b-sh lib-x)

build_PID_Package()
```

The `find_package` function is a standard cmake function used to find and configure other packages. In the example, we search for a package named "the-testpack-a" (that is also a PID package) with version compatible with 1.0 (for PID packages compatibility as the same meaning as in section 2.2.1, otherwise the meaning is package dependent). The package is REQUIRED meaning that it must be found. We specifically require the **components** named "lib-b-sh" and "lib-x". As experienced CMake users may notice there is no difference in the usage of the `find_package` function as regard of its standard use.

Then PID development process imposes to declare dependencies of the current package. Indeed this is not because you try to find other package that you will use them, even if obviously this assumption will be right most of time. A dependency simply means that components of the package are using components from other packages (see section 2 to understand what is a component). For instance, the current package uses the package "the-testpack-a" with minimum version 1.0. To make this declaration possible PID provides the `declare_PID_Package_Dependency` function. This function can be used in two different ways:

- it is used to specify a dependency to an **external package** (using the keyword EXTERNAL after the package name). An external package is just a *reference* associated to a path that points to the root folder of a **NON PID** package. This reference will be used later when defining components. A version information can be appended but is not used. A good way to do is to install external package into the **external** folder of the workspace, when possible, so that they will be managed by PID in a easier way. External package are packages installed by hand by the user in his file system, either in system or add-hoc folders. For packages that are supposed to be part of the operating system, there is no need to specify the dependency since this dependency is satisfied "by default" by the system.
- it is used to specify a dependency to a PID package (using PID keyword). Then version and component requirement informations can be used exactly as in the example. Relationship between PID packages are stronger since their discovering/install configuration will be done automatically.

Finally, the CMakeList.txt file call the `build_PID_Package` function. This line is mandatory in order to allow the build of the package: compilation, installation, deployment, API doc generation, etc. Without this call, the CMake process would simply do nothing.

5.1.3 Dealing with conditional dependencies

The previous example is quite simple since it directly deals with **required** dependencies. Nevertheless, when using cmake one sometimes have to deal with **conditional dependencies**. Indeed, **conditional dependencies** allow to configure the build according to the OS requirements or to the configuration of user's station. This conditional dependencies are managed according to the following pattern (or a derivative from this solution):

```
find_package(the-testpack-a)
if(the-testpack-a_FOUND)
    set(USE_the-testpack-a TRUE CACHE INTERNAL "" FORCE)
     declare_PID_Package_Dependency (PACKAGE the-testpack-a
                PID VERSION 1 0 COMPONENTS lib-b-sh lib-x)
else()
    set(USE_the-testpack-a FALSE CACHE INTERNAL "" FORCE)
    find_package(the-testpack-x)
    if(the-testpack-x_FOUND)
       set(USE-the_testpack-x TRUE CACHE INTERNAL "" FORCE)
       declare_PID_Package_Dependency (PACKAGE the-testpack-x
                PID VERSION 1 3 COMPONENTS lib-any)
    else()
       set(USE-the_testpack-x FALSE CACHE INTERNAL "" FORCE)
       MESSAGE("alternative system dependency not satisfied.
       Install/select either the-testpack-a or the-testpack-x")
       return()
    endif()
endif()
```

Conditional dependencies is used to describe dependencies that are optional or to describe an alternative requirement when multiple different packages can be used for the same purpose. This later case is shown in the previous code. Defining such conditional dependencies is made using classical CMake mechanism (using `find_package` and `option`/`set` commands) is combination with the `declare_PID_Package_Dependency` function previously explained. In the previous example, there is an implicit priority in the usage of required packages: "the-testpack-a" will be used prior to "the-testpack-x" if it is found. The CMakeList.txt file can also use the `option` command to let the user select its preferred package. The only mandatory descriptive elements are:

- if a package is to be used, **whether it has been previously found or not**, then the `declare_PID_Package_Dependency` function **must be called** to tell to PID system which package (PID or external) is required.

- Each possibly used package must be bound to a CMake **variable that indicates if the package is really used or not**. For instance see the variable `USE_the-testpack-a`. These variables will be used later, when declaring components, to know how to configure component dependencies according to required packages.

5.2 Defining Library components

Once package dependencies have been defined, the package developers can then define components of the package and their relationship with these dependencies. Most common components are library components : they are used by developers to define reusable functionalities. All libraries are defined in the CMakeList.txt file contained in the **src** folder of the package repository.

PID defines three types of libraries, matching the three classical types available in C/C++. It provides a systematic way to define these libraries and automate their use, avoiding the user to know precisely how to deal with each type and their relative properties. Following subsections explain how to declare each of these types. These definitions rely on cmake functions provided by PID: `declare_PID_Component` and `declare_PID_Component_Dependency` (see appendix 7).

5.2.1 Header libraries

Header libraries are not compiled to produce a binary object. They are just made of a set of header files. This kind of library used is often used for template libraries definitions, for instance the Boost library. Such a library is never used at link time (never linked to another library or executable using a linker) but only at compile time (when including header files). The definition of a header lib should look like:

```
declare_PID_Component(HEADER_LIB NAME lib-y DIRECTORY lib_y)
declare_PID_Component_Dependency(COMPONENT lib-y
               EXPORT DEPEND lib-x PACKAGE the-testpack-a
               EXPORTED_DEFINITIONS USE_EXTERNAL_LIB_X)
```

The first thing to do is to declare the header library by using the function `declare_PID_Component`:

- the HEADER_LIB keyword is used to declare a header library.
- the NAME keyword is used to define the identifier of the component in PID, whose unicity must be preserved. In the example the name is "lib-y".
- the DIRECTORY keyword is used to say in which sub-directory of the **include** folder the header files are found, in the example the "lib_y" sub-folder. The direct consequence is that all headers of a given library **must be placed in a unique folder**.

Then, depending on the library content, some dependencies can be attached to the library, using `declare_PID_Component_Dependency`. Indeed a header library can depend on other libraries, either header, static or shared:

- the COMPONENT keyword is used to specify for which component a dependency is defined, in the example the previously defined "lib-y" header library.
- the EXPORT keyword specifies that the lib-y component export the required dependency. Exporting means that the reference to the required component is defined in the interface of the library. Since a header lib is only made of an interface, **it must export each of its dependencies**.
- the DEPEND and PACKAGE keywords are used to specify the dependency: here the lib-y header library depends on the component "lib-x" of the PID package "the-testpack-a". Declaring an external or system dependency or even an internal dependency is slightly, but follows the same logic, see appendix 7.
- the EXPORTED_DEFINITIONS is used to specify values of C preprocessor definitions that are exported by the library. In the example the exported definition is `USE_EXTERNAL_LIB_X`. exported definition are used by components that will use lib-y to configure its code adequately. Since a header lib is only made of an interface, all its definitions must be exported and it can have no internal definitions.

On interesting property of PID is to be able to declare different components from the same code. For instance:

```
declare_PID_Component(HEADER_LIB NAME lib-y-bis DIRECTORY lib_y
                      EXPORTED_DEFINITIONS NUSE_EXTERNAL_LIB_X)
```

In this later example, a new component named "lib-y-bis" is created from the same source code contained in the lib_y folder. The differences with the previous component are that the "lib-y-bis" has no dependencies and it undefines (using N in front of the preprocessor definition) the definition `USE_EXTERNAL_LIB_X`. This is useful to declare many alternatives from the same code.

5.2.2 Static libraries

Static libraries are binary archives that define some functionalities. A static library is made of:

- a set of header files that define its interface (i.e. what is available for library users).
- a set of (compiled) binary objects that implement its behaviour.

Its interface is used at compile time (when its header are included) and its contained objects are linked to executables and shared libraries at link time, so they no more exist at run time. The definition of a static lib should look like:

```
declare_PID_Component(STATIC_LIB NAME lib-c-st DIRECTORY lib_c
                  INTERNAL DEFINITIONS NA_VERY_SPECIFIC_IMPLEM)
declare_PID_Component_Dependency(COMPONENT lib-c-st
                  EXTERNAL Boost INCLUDE_DIRS <Boost>/include)
declare_PID_Component_Dependency(COMPONENT lib-c-st
                  EXPORT DEPEND lib-y-bis)
```

As for any component, the first thing to do is to declare by using the function `declare_PID_Component`:

- the STATIC_LIB keyword is used to declare a static library.
- the NAME keyword is used to define the identifier of the of the static library, "lib-c-st" in the example.
- the DIRECTORY keyword is used to say in which sub-directory of the **include** folder the header files of the static library are found, in the example the "lib_c" sub-folder. The **same folder name** is used to specify in which subdirectory of the **src** folder the source and non-public header files of the library are found. For a same library, this constraints the user to use same folder names between **include** and **src** directories.
- the INTERNAL DEFINITIONS is used to specify definitions that affect only the implementation (i.e. that is not used in any header file of the library). In the example the "lib-c-st" undefines the preprocessor definition A_VERY_SPECIFIC_IMPLEM .

As readers can notice, the declaration is quite the same as for header libraries. Note also that static libraries can define exported definitions (as header libraries) for those which are used in their header files. The declaration of dependencies also follows the exact same pattern. In the example:

- "lib-c-st" static library is using an external package named "Boost". As boost is a pure header library it only needs to specify where to find its header files, using the INCLUDE_DIRS keyword. The include path specified is relative to the Boost package root folder (using the <Boost> specifier).
- "lib-c-st" is also using (specified with keyword DEPEND) another library "lib-y-bis" that is defined in the same package (since no PACKAGE keyword is used). It exports "lib-y-bis" meaning that its header files contain include directive over header files of the "lib-y-bis".

5.2.3 Shared libraries

Shared libraries are binary objects that define some functionalities. A shared library is made of:

- a set of header files that define its interface (i.e. what is available for library users).
- a binary object (.so on linux) that implements its behaviour. This

Its interface is used at compile time (when including its headers) and its binary object checked at link time and truly used at run time, either when the executable using it is loaded or when it explicitly load it at run time. The definition of a shared lib is more or less the same as for static libraries and should look like:

```
declare_PID_Component(SHARED_LIB NAME lib-c-sh DIRECTORY lib_c
                    INTERNAL DEFINITIONS A_VERY_SPECIFIC_IMPLEM)
declare_PID_Component_Dependency(COMPONENT lib-c-sh
                    EXTERNAL Boost INCLUDE_DIRS <Boost>/include)
declare_PID_Component_Dependency(COMPONENT lib-c-sh
                    EXPORT DEPEND lib-y)
```

In the previous example, the function `declare_PID_Component` is used in a common way:

- the SHARED_LIB keyword is used to declare a shared library.
- the NAME keyword is used to declare "lib-c-sh".
- the DIRECTORY keyword is used to define **include** and **src** sub folders where to find code. In the example reader can notice that the shared library "lib-c-sh" is built from the same code as static library "lib-c-st".
- the INTERNAL DEFINITIONS is used to specify to define the preprocessor definition A_VERY_SPECIFIC_IMPLEM contrarily to "lib-c-st".

This example shows how shared and static libraries can be built from the same source code and how developers can define alternative implementation for part of their code using preprocessor definitions. Their dependencies can also vary:

- "lib-c-sh" shared library is using the Boost external package the same way as "lib-c-st".
- "lib-c-sh" is using (specified with keyword DEPEND) the library "lib-y" (that is defined in the same package) instead of "lib-y-bis" used by "lib-c-st".

5.3 Defining Application components

In order to produce programs usable by end-user package can also contain application components. Application components that are designed to be used by end-users are defined in the CMakeList.txt file contained in the **apps** folder of the package repository. Test applications are specifically used to test package libraries or applications and are placed in the **test** folder of the package repository.

PID defines three types of applications explained in following subsections. These definitions rely on same cmake functions already presented in libraries component section (see also appendix 7).

5.3.1 Standard applications

By standard application we mean application that are to be used by end-users of the package, or a run-time software component that can be deployed using a middleware. The definition of a standard application should look like:

```
declare_PID_Component(APPLICATION NAME app-b1 DIRECTORY app_B1
                INTERNAL INCLUDE_DIRS common_defs common_types)
declare_PID_Component_Dependency(COMPONENT app-b1
                DEPEND lib-b-sh PACKAGE the-testpack-a)
```

As for library components (see 5.2), the first thing to do is to declare the application by using the function `declare_PID_Component`:

- the APPLICATION keyword is used to declare a standard application.
- the NAME keyword is used to define the unique identifier of application, "app-b1" in the example.
- the DIRECTORY keyword is used to say in which sub-directory of the **app** folder the source files of the application are found, in the example the "app_B1" sub-folder.
- the INTERNAL INCLUDE_DIRS allows to specify additional directories (sub folders of the **apps** folder) where to find non-public header files, in the examples folders "common_defs" and "common_types".

Then developers can add dependencies for the application, exactly the same way as for libraries using the `declare_PID_Component_Dependency` function:

- the COMPONENT keyword is used to specify the application that declares a dependency, "app-b1" in the example.
- DEPEND and PACKAGE keyword are used to target a specific component from another package, here the shared library "lib-b-sh" of the package "the-testpack-a". Dependencies management work the same way as for libraries.

5.3.2 Example applications

Example applications are little pieces of executable code whose only purpose is to provide to developers simple way of using libraries defined in the package. The definition of an example application should look like:

```
declare_PID_Component(EXAMPLE_APPLICATION NAME ex-b DIRECTORY ex_b)
declare_PID_Component_Dependency(COMPONENT ex-b DEPEND lib-b-sh )
```

From a strict C/C++ point of view example application are just like standard application, in other word an executable binary object. From PID point of view this is also nearly the same:

- Example application are developed with same rules as standard application except that we have to use the EXAMPLE_APPLICATION keyword within `declare_PID_Component` function (see previous code).
- Developers can decide (using dedicated CMake option in cache) to avoid compiling example applications since most of time they are not really useful.
- Example application code is integrated into the API documentation.

5.3.3 Test units

The **test** subdirectory contains a CMakeList.txt file that builds test units. The organization into subdirectories follows the same logic as for libraries and applications. Test units are specific components because they will not be installed with package binary. They are just used to test the validity of the libraries code, for instance to be sure it behaves properly or respects some backward compatibility constraints.

The first step when playing test is to define test applications, doing something like this in the CMakeList.txt file of the **test** folder:

```
declare_PID_Component(TEST_APPLICATION NAME test-b
                                       DIRECTORY test_b)
declare_PID_Component_Dependency(COMPONENT test-b
                                 DEPEND lib-b-sh )
```

This "test-b" application is in charge of testing the library "lib-b-sh". Then, always in the CMakeList.txt file of the **test** folder, this test application can be used to launch series of tests, using standard CTest tooling integrated in CMake:

```
add_test(correctness_of_lib-b-sh_step1 test-b "first"
                                              "124" "12")
add_test(correctness_of_lib-b-sh_step2 test-b "second"
                                              "12" "46")
add_test(correctness_of_lib-b-sh_step3 test-b "second"
                                              "0" "87")
```

In the previous example, one can see that the same test application "test-b" may be used to run series of test, simply by changing its input parameters. Of course different test applications may be used to test same libraries in needed (like "test-b-back" used to test backward compatibility of "lib-b-sh". Another option is to use generic software test tools or framework (e.g. Valgrind, C-Cover, Purify, etc.) to check for validity of general properties (e.g. runtime memory errors, code coverage, analysis of code metrics), but this is beyond the topic of this document.

A simple and standard way to proceed is to define test application that take different arguments:

- an argument represent the tested functionality (e.g. "first" in the previous example). A functionality can be seen as a particular use of the test library's API in order to obtain a given behaviour. It is implemented as a given block of code inside the test application.
- one or more arguments represent input parameters (e.g. "124" in first test).
- one or more arguments represent the expected output parameters (e.g. "12" in first test).
- the test program (e.g. "test-b") call the adequate target functionality (e.g. "first") of the tested library (e.g. lib-b-sh) with adequate input parameters (e.g. "124") and check if the result is the expected one (e.g. "12"). If **successful it returns 0**, otherwise it returns an error code (something else than 0).

The previous code will automatically generate a sequence of tests whose result is **PASSED** or **FAILED** according to the result returned by the test program. Tests failure are not blocking the whole build and install process BUT developers should take care to validate all tests before releasing a version of the package.

5.4 Generating API documentation

When a library is defined, it is intended to be used by third party developers. To this end, it is always useful to have a clear way to consult the API provided by this library. The api documentation has to be as close as possible to the source code, that is why the best way is to use an api documentation generation tool like **doxygen** and to automate its use directly during the build process. We choose **doxygen** because it is a cross-platform tool very well suited for C/C++ code.

PID automatically manage the generation of API documentation with doxygen. The generated doc is installed in the binary package version folder in the share/doc/html sub-folder. If latex is installed on the building host, it is also possible to generate an equivalent pdf document that will placed in the share/doc/latex sub-folder. The developers of a package can configure this generation in two ways, presented in following subsections.

5.4.1 Documenting headers

The API documentation requires that the users document the header files contained in each sub-folder of the **include**. The way to document headers is defined by doxygen tooling. Generally speaking, it consists in defining comments with specific format in header file code. As an example, the comment at the beginning of a header file should look like this:

```
/**
 * @file MPO700interface.h
 * @author Robin Passama
 * @brief interface of the neobotix communication library.
 * @example pc_side_simple_interface.c
 * Created on June 2013 18.
 * License : CeCILL-C.
 */

#ifndef MPO700_INTERFACE_H
#define MPO700_INTERFACE_H
...
```

In this example are specified general information about the header:

- the `@file` property specifies the name of the file.
- the `@author` property gives names of authors of the file.
- the `@brief` property gives a quick description of the header purpose.
- the `@example` property allows to specify a source code giving an example on how to use the API defined below. This property should refer to the source code of example applications.

Then developers have to document each of their functions/methods and classes/structures/enumerations by putting a comment before their declaration, the same way as:

```
/**
* Function used to initialize the communication interface with
* the robot (must be called before any other call).
* By default the robot is in monitor mode just after this call.
* @brief initialize communication with robot.
* @param [in] if_name is the name of the eternet interface used
* for communication (e.g. eth0).
* @param [in] mpo700_ip is the IPV4 address of the robot.
* @return 0 if initialization failed, 1 otherwise
*/
int init_MPO700_Robot(char* if_name, char * mpo700_ip);
...
```

This example is the way a function named `init_MPO700_Robot` is documented with doxygen:

- the `@brief` property gives a quick description of the function utility.
- the `@param` property documents each argument of the function.
- the `@retuen` property documents the return valeu of the function.

For a more detailed explanation, readers should refer to the doxygen tool tutorials that can be found online.

5.4.2 Adding some content by modifying Doxygen configuration file

When developers have documented their header files, they have to do nothing more to get a standard html or pdf document. If they want to customize these documents, they have to:

- add some content in the **doc** sub-folder of the package's **share** folder. For instance a set of images can be put in a **img** sub-folder of the **doc** folder.
- modify the doxygen configuration file ("Doxyfile.in") that can be found in the **doxygen** sub-folder of the package's **share** folder. This file is used by doxygen to know how to generate the documentation. For instance, one can modify the IMAGE_PATH contained in this file so that it points to the new **img** folder.
- Then images can be referenced directly into doxygen headers comments using a specific keyword (@image).

Configuring doxygen behaviour is far beyond the scope of this document, interested readers may refer to online documentation and tutorials on the doxygen tooling. The only thing that is absolutely require is to let some variables of the "Doxyfile.in" unchanged: all variables whose value is **surrounded by the @ symbol must be let unchanged**. These variables are automatically fill by PID cmake scripts, for instance:

```
...
# The PROJECT_NAME tag is a single word
PROJECT_NAME           = "@DOXYFILE_PROJECT_NAME@"

# The PROJECT_NUMBER tag can be used to enter a version.
PROJECT_NUMBER         = "@DOXYFILE_PROJECT_VERSION@"

# The OUTPUT_DIRECTORY tag is used to specify the (relative or
# absolute) base path where the generated documentation will
# be put.
OUTPUT_DIRECTORY       = "@DOXYFILE_OUTPUT_DIR@"
...
# If the GENERATE_HTML tag is set to YES (the default) Doxygen
# will generate HTML output.
GENERATE_HTML          = @DOXYFILE_GENERATE_HTML@

# The HTML_OUTPUT tag is used to specify where the HTML docs
# will be put.
HTML_OUTPUT            = "@DOXYFILE_HTML_DIR@"
```

When the doxygen configuration in generated by cmake, this later use the Doxyfile.in pattern file and automatically fills all fields surrounded by the @ symbol in "Doxyfile.in" according to corresponding cmake variables. Modifying these field would provoke bad behaviours.

5.4.3 CMakeList.txt of the share folder

The CMakeList.txt file of the share folder does not explicitly manage installation of the API documentation. Nevertheless, if developers add resources to the **share** folder like for instance images, these resources may be needed when the package binary is installed. In such a case the CMakeList.txt has to manage the installation of these resources, using the classical CMake `install` command. These resources have to be placed in the binary package's **share** folder with a command like:

```
install(DIRECTORY img
        DESTINATION ${${PROJECT_NAME}_INSTALL_SHARE_PATH})
```

This later command will install the img folder (that is in the share folder) and all its content into the adequate share folder of the installed binary package. The same process can be used for instance for documents like README/INSTALL files, using the `install(FILE )` command.

5.5 Configuring the package

PID packages provide a set of generic variables to allow the configuration of the build/install process. The configuration of these CMake cache variable is made using `ccmake ..` command in the **build** directory of the package and then by using Cmake configuration interface.

- BUILD_AND_RUN_TESTS (default to OFF): If this option is ON the CMake process will build test applications and run unit tests that have been defined in the CMakeList.txt file of the **test** folder.
- BUILD_API_DOC (default to ON): If this option is ON, the CMake process will build the html API documentation of the package.
- BUILD_LATEX_API_DOC (default to OFF): If this option is ON and if BUILD_API_DOC is ON, the CMake process will build the pdf document containing the API documentation of the package.
- BUILD_EXAMPLES (default to ON): If this option is ON, example application will be build and installed in the binary package resulting from the build process.
- BUILD_PACKAGE_REFERENCE (default to OFF): If this option is ON, the CMake process will generate a reference file for the package. This reference file is installed in a dedicated folder of the containing workspace that contains a reference file for each known PID package. This file be used for the management of package downloading and deployment (either for source repository and binary archives).
- BUILD_WITH_PRINT_MESSAGES (default to OFF): If this option is ON, the preprocessor definition PRINT_MESSAGES will be set. It should be used by developers each time they want to log information from their running code.

- USE_LOCAL_DEPLOYMENT (default to ON): If this option is ON, the package binary version resulting from the build process will be installed in a version folder with the form own-<version number> instead of a folder with <version number> form. This is useful for developers to precisely control which version of the code he uses. This option is used when working on multiple packages at the same time and when user want to target its own developed binary packages instead of already released ones.

- GENERATE_INSTALLER (default to OFF): If this option is ON and USE_LOCAL_DEPLOYMENT is OFF, the CMake process will generate a "ready to install" relocatable binary archive of the package version that will be built.

- REQUIRED_PACKAGES_AUTOMATIC_DOWNLOAD (default to OFF): If this option is ON the CMake process will automatically try to install adequate binary package version when they cannot be found in the local workspace. The "automatic download" procedure will do one of the following action:

  - if the required package repository exists in the workspace, CMake will retrieve the adequate Git tag corresponding to the version to install, go to the corresponding commit and build the binary package for this commit.
  - If the required package repository does not exist, CMake will use **package reference files** contained in the workspace in order to know where to find package on the internet. Then depending on the situation:
    - CMake will download a binary package version archive that is 1) available on internet, 2) adequate regarding version constraints. After download, CMake will install the archive.
    - if no adequate binary package version archive is available, then CMake will clone the git repository (if the user has access to it) and then do the same as the first possible action.
    - if the user has no access to the package source repository, then the build/install process will stop on an error.

5.6 Controlling package build process

The package build process is controlled with native build tools. For now, only Linux is supported, so examples are provided considering the Makefile build control tool. All build related commands used must be called from the **build** folder of the package.

As shown in section 5.5 CMake configuration tool is used to configure package:

```
ccmake ..
```

Then Each time a file or directory is added to the package CMake is used to reference it or each time any CMakeList.txt file of the package is modified:

```
cmake ..
```

Once this last command has been executed, developers can use native build tool to build the package. PID system defines a set of targets that can be used:

- `make` compiles and links the source code.
- `make test` run tests (see 5.3.3). Test units must have been compile first. This command is available only if BUILD_AND_RUN_TESTS option has been set to ON.
- `make doc` generates API documentation (see 5.4). This command is available only if BUILD_API_DOC option has been set to ON. If BUILD_LATEX_API_DOC option has been set to ON, the pdf document of the API is generated when running the command.
- `make install` installs the package binary version resulting from the build into the adequate folder of the workspace (see section 3.1.2).
- `make package` generates the binary package version relocatable archive (using the CPack tool). This command is available only if the GENERATE_INSTALLER option has been set to ON and USE_LOCAL_DEPLOYMENT option has been set to OFF.
- `make package_install` install the binary package version relocatable archive in the adequate **installers** folder of the workspace (see section 3.1.2).
- `make build` runs all the previous commands sequentially in the adequate order (same order as items of this list) and according to options selected by the developer.

Important: The build process takes place in release and debug mode in the same time: the **release** sub-folder of the **build** folder contains build artefacts for the *Release* configuration mode and the **debug** sub-folder contains build artefacts for the *Debug* configuration mode. Developers don't have to worry about these directories they are automatically created and managed by PID system. The CMake cache variables of the package (see section 5.5) are the same for both configuration mode. Nevertheless, depending on the configuration mode, dependencies, notably those to external packages, can be different and so some dedicated CMake variables can appear in the CMake cache. The only think to understand is that variable with `_DEBUG` appended are relative to *Debug* mode otherwise they are relative to *Release* mode.

Important: The CMakeList.txt files of the package can do different things according to the configuration mode using the CMAKE_BUILD_TYPE variable to check built mode as in any CMake script. Nevertheless there are some rules to respect in order to make PID work correctly:

- All components defined in *Release* mode MUST BE DEFINED in *Debug* mode and reversely. In other words, both modes define the same components for the package (i.e. with same names, using the same `declare_PID_Component` function).
- Components and package dependencies can be different between modes, nevertheless the developer should always keep in mind that debug version of a component should reflect as close as possible the release version. Unless the contrary is absolutely mandatory, dependencies to PID packages and PID components should be the same in both modes, only external dependencies should change.

5.7 Resulting binary package version

From the complete build/install process (see section 5.6) a binary package version is generated and installed into the adequate folder of the workspace (see section 3.1.2).

The install process is managed by PID system so developer should not worry about how it takes place. The only exception is for documents and other resources (like images) placed into the source package repository's **share** folder that are installed by hand by developers (see section 5.4.3).

Figure 6 provides an example of the workspace's install folder containing two installed packages. There can be many versions of the same package installed in the workspace as for the package "the-testpack-c", which has two

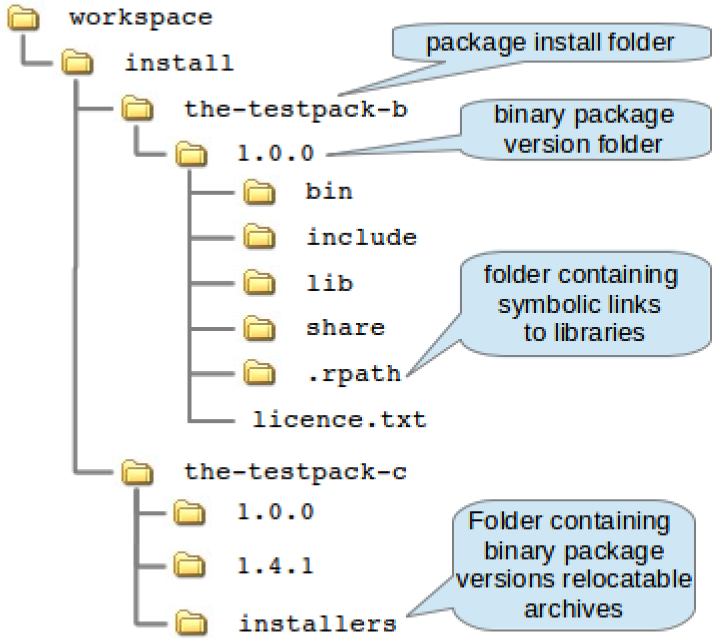

Figure 6: Example folder hierarchy for installed packages

versions installed. The **installers** folder for this later package may contains many binary package version relocatable archive (quickly called **package installer**), for instance two for each installed version. Indeed each binary version is associated with two **package installers**: one for *Release* mode and one for *Debug* mode. **Package installers** are zip archives (since cmake is able to compress and extract those archives in a cross-platform way) with following pattern for their name: <package name>-<package version number>-<operating system>[-dbg].zip. **Package installers** for *Debug* mode have an additional "-dbg" postfix notation appended to their name.

Each binary package version folder also have a `.rpath` folder which is a PID specific folder used for configuring runtime links of PID components. It contains, for each binary executable component (share libraries binaries, applications binaries) of the package a set of symbolic links that points to the adequate shared libraries. PID system can reconfigure "on demand" run-time dependencies depending on the shared libraries versions installed and used. This system allows to create relocatable and reconfigurable binary code without using system mechanisms. Developers should never change the content of the **.rpath** folder "by hand", otherwise it could cause troubles.

6 Good practices

6.1 Structuring development with packages

6.1.1 Overview

The basic guideline is to separate software into many packages that are structured according to a strict "depends" hierarchy as described in figure 7. The "depends" relationship simply describes package dependencies as they are described in the CMakeList.txt of packages using the CMake PID function `declare_PID_Package_Dependency` (see section 5.1.2). It just means that a source package requires another package to be installed in the workspace or in the system (for external packages).

From a functional point of view a dependency can generally represent one of these two alternatives relationship:

- an **extension** relation : a child package **extends** a base package if it provides some *functionalities* that specialize/extends those of the base package. This is a typical relationship when a library extends the class hierarchy provided by a library of the parent package, or/and when a more specialized/complete version of an application is provided.

- a **use** relation : a child package **uses** a base package if it provides *new functionalities* built onto those provided by the base package. This is the case when new libraries are using more basic ones or when new applications are built using existing components.

6.1.2 General guideline applied to packages

- The names of packages **must be unique** considering all developed packages in the frame of the laboratory. Names are alphanumeric lower case ascii characters, without space but can include '-' for separating names in compound names.

- **Cyclic dependencies between packages are forbidden**.

- If a package has some dependencies induced by lower levels of the class hierarchy, prefer making a package for the higher level with less dependencies, and one or more dependent **extension** packages for the lower levels, each of them with the strict minimum required dependencies.

- When developing a component for a given middleware (ROS, OROCOS, etc.) always put the functional code (library) in one package and extend this package each time a component is built from this code.

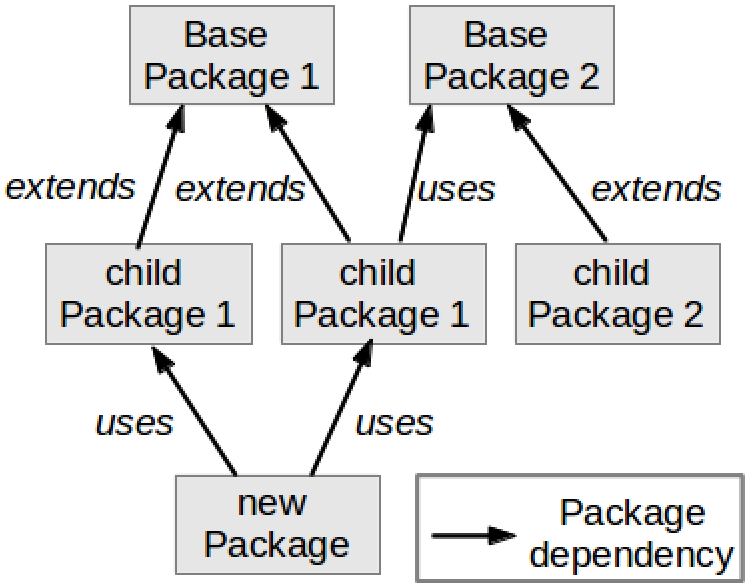

Figure 7: An example hierarchy of packages

- Dependencies to multiple packages are allowed but the developer should keep in mind to have the lowest possible number of dependencies for a given package, to keep relationship between packages understandable.

- The number of **direct dependencies to external packages** should be very limited for a given package (0 to 5 max).

- If possible, it is a better choice to choose a **commonly used external package** (e.g. boost, eigen, etc.) than a very specific one to define a dependency with a required external package.

6.1.3 Guideline relative to developed functionalities

- Try limiting the number of software artefacts generated by a package. Typical use is **one package per functionality**. For a given *functionality*, the package defines :

  - libraries implementing the functionality core behaviours and eventually example applications to explain how to use these libraries.
  - applications if this functionality can be directly used by end-users.
  - test units to control that all these libraries and components work well.

- Different variations of a same *functionality* should be developed in the same package. For instance if developers define many variations of a same library (e.g. typically static and shared version) that all implements the same *functionality* (i.e. do the same thing from user point of view), each variation should be contained as a specific component in the same package.

- Components participating to the same *functionality* should be developed in the same package. For instance if a set of application components (e.g. command line tools) are used to achieve a same objective, they should be developed in the same package. As a concrete example, the CMake command line tools suite could be developed in the same package.

- Limit the number of hierarchical level to the strict necessary. As a global guideline we discourage having more than 4 hierarchical levels for the "extends" dependency to keep the extension hierarchy understandable.

- A given package should only have an "extends" dependency with one package at maximum.

6.2 Conventions

This section explains good practices and guidelines when developing C/C++ code in PID packages.

6.2.1 Files and folders naming convention

- Package's folder names have the same name as the repository which is also the same name as the one declared in the CMakeList.txt file of the package (see section 5.1).

- Sub-folders of package's standard folder (**src**, **include**, **apps**, **test**) are alphanumeric lower case ascii characters, without space but can include underscores ('_') for separating names in compound names.

- All executables and libraries names are alphanumeric sequence of lower case ascii characters words using only '-' as special characters between each word.

- C/C++ files' names are alphanumeric sequence of ascii lower case words using only underscores ('_') as special characters between each word.

- C++ source files have a .cpp or.cc extension , C source files have a .c extension.
- C++ header files have a .h or.hh, C header files have a .h extension.
- Template header files have a .hpp extension and template functions/classes should always be entirely defined in header files, not in source files.
- Files defining a C++ class have the name of the class. Only one class can be defined in a file (except internal classes).

6.2.2 C/C++ objects naming convention

- Classes/structures/enums/namespaces: their name is made of words, each word starting with a upper case character and other characters are lower case. Avoid too long names (e.g. `class FooBar` NOT `class TheClassFooBarIsAReallyLongNameForAClass`).
- Names of variables and functions' arguments are alphanumeric lower case sequence of words using only underscores ('_') as special characters between each word (e.g. `int my_variable_name`).
- Names of classes' attributes follows the same rule as variables but ends with an underscore (e.g. `int my_attribute_name_`).
- Constants are all capital, words are separated by underscores (e.g. const int MY_CONSTANT).
- Names of functions/methods have lower case characters except the first character of each word, but not for the first word that is full lowercase. Words are separated by underscores (e.g. `int this_Is_The_Name_Of_My_Function(...)`).

6.2.3 C/C++ coding guideline

- **Indentation**: the indentation of the code should follow the bloc structure of the program. Put only one instruction by line.
- **Comprehensive naming** : make your variables/functions/classes' names explicit and comprehensive as regard of the context. Use real life words, avoid meaningless random sequence of letters (e.g. one-letter names) except for integral iterator variables (i, j, k). An exception is tolerated if the notation come from a paper (e.g. matrix name).

- **Package scoping**: use namespaces to scope your code and gather all classes, functions and global variables (e.g. constants) that belong to a same package. Each package as thus its own namespace whose name is the name of package (same words) but with namespace naming convention.

- **Namespace inheritance**: when a package defines an "extends" dependency with another package, its namespace is prefixed with its parent package namespace (e.g. namespace `parent` for parent package namespace, namespace `parent::child` for the child package namespace).

- **Class inheritance**: The basic usage is to subclass by using the public virtual statement (this corresponds to the standard specialization mechanism when using multiple inheritance in object oriented programming. Never use protected or private inheritance. Use non virtual inheritance only if you are sure that multiple inheritance will not take place in the hierarchy of the target class.

- **Class protection**: all attributes of a class must be `private` : define `public` or `protected` getters and/or setters only for attributes that can be accessed outside the class. Methods used only internally to a class must be `private`. Methods that are used in base class but that can be redefined in child classes must be`protected`. In a sub-class, never change the protection of elements defined in its base class.

- **Friendship**: use `friend` statement with care: `friend` keyword should be used only if a class as specific access to a "private interface" (set of private or protected methods) of another class (the accessed class declares the accessing class as a friend). This is used when two classes in two distinct inheritance hierarchies are deeply bound to each other.

- **Polymorphism**: by default all `public` and `protected` methods must be `virtual`, and `private` methods have to be not `virtual`. If a `public` or `protected` method cannot be redefine in sub-classes then make them not `virtual` (e.g. most of time the case for getters/setters).

- **Templates**: templates classes and functions must be completely defined in header files.

- **Inlining**: never inline a `virtual` method ; inline only short methods that are to be widely used (getters and setters for example).

- **Guarding headers**: the headers must be protected by multiple inclusion by using `#ifndef` guards. Guards names follow constants naming convention with following structure : <package>_<file name>_H. For instance:

```
#ifndef PACKAGE_FILE_H
#define PACKAGE_FILE_H
...
#endif //PACKAGE_PATH_FILE_H
```

- **Functions**: output arguments to methods/functions (i.e., variables that the function can modify) are passed by pointer, not by reference. Input "heavy objects" should be passed by const reference or pointer to `const` objects. For instance:

```
int example_Method(const Foo & in, Bar* out);
int other_Method(const Foo * in, Bar* out);
```

- **Functions arguments**: limit the number of arguments of a function to 5 at maximum.

- **Be a const addict**: add `const` whenever it is possible: in the parameters of a function (input/output) and for methods that must not modify the class attributes. Return elements by `const` reference when you want to make readable an object attribute of a class or by pointer on `const` object when you want to make readable a pointer attribute.For instance:

```
const std::vector<double> & get_Attribute_Value() const;
const Bar * get_Attribute_Pointer() const;
```

- **Preprocessor macros and constants**: should be avoided. Prefer `inline` functions, `enums`, and `const` variables.

- **Preprocessor instructions**: usage of #ifndef, #if, etc. should be limited as far as possible, except when dealing with platform specific code (OS specific libraries) or when dealing with alternatively available package dependencies (see section 5.1.3).

- **Exceptions**: exceptions are the preferred error-reporting mechanism for C++, as opposed to returning integer error codes. Always document what exceptions can be thrown by your package, on each function /

method. Do not throw exceptions from destructors. Do not throw exceptions from callbacks that you do not invoke directly. When your code can be interrupted by exceptions, you must ensure that resources you hold will be deallocated when stack variables go out of scope. In particular, mutexes must be released, and heap-allocated memory must be freed. Accomplish this safety by using smart pointers.

- **Documentation**: use doxygen annotations in header files to describe all methods (arguments, return value, usage, etc.), attributes, class (role and relationship with other classes), etc. These comments will be automatically included when the api documentation will be generated (see section 5.4.1).

- **Licensing**: systematically apply a license, authors and version information at the beginning of all source file (C/C++ headers and source). For license text, see the license cmake script file corresponding to the license used (it can be found in the workspace, see section 3.1.1).

- **Threading**: avoid the management of multi-threading (processes/thread execution management, usage of IPC and synchronization mechanisms) into libraries and implement these mechanisms inside applications. Library should contain only computational code while applications embed control-flow management code. This rule is not valid for libraries, like those of middleware, implementing task management and communication mechanisms.

Conclusion

Concerning package development, developers have to refer to dedicated document explaining coding rules and general code development guidelines.

Appendix

7 PID cmake functions API

With PID there are few functions to use to precisely declare the content of a package. Each of these functions is detailed below. Note that optional arguments are signalled using the optional keyword into brackets.

7.1 declare_PID_Package

Inputs argument:

- AUTHOR <name>. This is a string defining the name(s) of the reference author.
- {optional} INSTITUTION <institutions>. This is a string defining the institutions to which the main author belongs.
- YEAR <dates>. This is a string character that reflects the life time of the package, for instance "year of creation - year of last modification".
- LICENSE <license name>. This is the name of the license applying to the package. This license name must be defined by a corresponding cmake file in the **licenses** directory of the workspace.
- {optional} ADDRESS <url>. This argument represents the url of the package's git repository. It can be let undefined for pure local repository but must be set once the package is delivered.
- DESCRIPTION <description>. This argument is a string that provides a short description of the package usage and utility.

Constraints: this function must be called in the root CMakeList.txt file of the package, before any other call to PID or `find_package` (or equivalent) functions.
Effect: initialization of package's internal state.

7.2 set_PID_Package_Version

Inputs argument:

- <major>. This is a positive digit indicating major version number.

- <minor>. This is a positive digit indicating minor version number.

- {optional} <patch>. This is a positive digit indicating patch version number. If not defined the patch version number of the package will be set to 0.

Constraints: this function must be called in the root CMakeList.txt file of the package, after `declare_PID_Package` and before `build_PID_package`.
Effect: set the current version number of the package. This will impact on the installed binary folder and configuration files.

7.3 add_PID_Package_Author

Inputs argument:

- AUTHOR <name>. This is a string defining the name(s) of an author of the package.

- {optional} INSTITUTION <name>. This is a string defining the names of the institutions to which the author belongs.

Constraints: this function must be called in the root CMakeList.txt file of the package, after `declare_PID_Package` and before `build_PID_package`.
Effect: add another author to the list of authors of the package.

7.4 add_PID_Package_Reference

Inputs argument: 2 signatures whether the user wants to add a binary package reference or to shadow the official git repository address.

- SOURCE option is used when user wants to shadow the official source repository address.

  - URL <address>. This argument specifies the address of the **private git repository** to use instead of the official address.

- BINARY option is used to add a reference to a binary package version.

  - VERSION <major> <minor> {optional}<patch>. This argument describes the full version number of the referenced binary package. <major> <minor> and <patch> are positive digits. If patch number is omitted then patch number is considered as 0.
  - SYSTEM <name>. This is the name of the target operating system. For now only linux supported.

– URL <url-rel> <url-dbg>. The two values of the URL arguments are: the url of the package binary in release version and the url of the package binary in debug version.

Constraints: this function must be called in the root CMakeList.txt file of the package, after `declare_PID_Package` and before `build_PID_package`.
Effect: reference addresses where to find a given binary package version for a given operating system. This information will be used to generate a CMake configuration file that will be used for retrieving adequate package versions.

7.5 declare_PID_Package_Dependency

Inputs argument: 2 signatures whether the dependency is an external or a PID package.

- For external packages:
  - PACKAGE <name>. The string without white space defining the unique identifier of the required external package.
  - EXTERNAL <path>. This is the path to the root directory of the external package. This path will be used as the basic reference for any folder/component of the external package that is referenced by local components.
  - {optional} VERSION <version string>. This version is a string without white space and following dotted notation of versions, that indicates which version of the external package is required.
- For PID packages:
  - PACKAGE <name>. The string without white space defining the unique identifier of the required PID package.
  - PID. This option is just used to indicates that the required package is a PID package.
  - {optional} VERSION <major> <minor>. Major and minor version number are positive digits that are used to constraint the required package to a given version.
  - {optional} EXACT. This option is used only if the VERSION argument is used. It indicates that the required package version is exactly the major.minor version required. Patch version is let undetermined.

– {optional} COMPONENTS <list of components>. This argument is used to specify which components of the required package will be used by local components. If not specified there will be no check for the presence of specific components in the required package.

Constraints: this function must be called in the root CMakeList.txt file of the package, after `declare_PID_Package`, after the `find_package` call to the same required package and before `build_PID_package`.
Effect: this function will register the target package as a dependency of the current package. These informations will be useful to generate adequate configuration files for the current package. Each required package **MUST BE referenced** using this function.

7.6 build_PID_Package

Inputs argument: none.
Constraints: this function must be the last one called in the root CMakeList.txt file of the package.
Effect: this function generates configuration files, manage the generation of the global build/install process and will call CMakeList.txt files of the following folders: **src**, **apps**, **test**, **share**.

7.7 declare_PID_Component

Inputs argument:

- <type>. This argument specifies the the type of the component. Its value is either STATIC_LIB (static library), SHARED_LIB(shared library), HEADER_LIB (header library), APPLICATION (stadard applciation),EXAMPLE_APPLICATION (example code), TEST_APPLICATION (test unit)
- NAME <name>. This is the string without white space defining the unique identifier of the component.
- DIRECTORY <dir>. This is the directory where to find component source. Depending on its <type> this directory is a sub-folder of the **src** (library), **apps** (example or standard application) or **test** (test unit) package's folders.
- {optional} INTERNAL. This argument is followed by other arguments that are only local to the component, i.e. that it does not export.

– {optional} DEFINITIONS <defs>. These are the preprocessor definitions internally used by the component's source code. For libraries these definitions must not be part of one of their header files. This is used to provide a given implementation to the component among different alternative, for instance to match operating system requirements.
– {optional} INCLUDE_DIRS <dirs>. These are additional directory where to find header files (that are not part of their interface for libraries).
– {optional} LINKS <links>. These are link flags, like path to lirabries. This argument is used mainly by applications.

- {optional} EXPORTED_DEFINITIONS <defs>. These are the preprocessor definitions that are contained in one or more of libraries header files. Applications do not export any definition.

Constraints: Depending of the <type> of the component this function must be called in the adequate CMakeList.txt file (either in the **src**, **apps** or **test** folders. It must be called before any call to declare_PID_Component_Dependency applied to the same declared component
Effect: this function is used to define a component in the current package. This will create adequate targets to build the component binary (if any) and install it.

7.8 declare_PID_Component_Dependency

Inputs argument:2 signatures whether the dependency is an external or a PID package.

- common arguments for both signature:
  – COMPONENT <name>. This is the string without white space that defines the local component for which a dependency is defined.
  – {optional} EXPORT. This is an option indicating if the component <name> exports the required dependency. Exporting means that the reference to the dependency is contained in its interface (header files). This can be only the case for libraries, not for applications.
  – {optional} INTERNAL_DEFINITIONS <defs>. These are preprocessor definitions used internally to the component source code,

when using the dependency. The definitions must not be part of the component header files (they are never exported).

  - {optional} IMPORTED_DEFINITIONS <defs>. These are preprocessor definitions **contained in the interface (header files) of the dependency** that are set (or unset) when the component use this dependency.
  - {optional} EXPORTED_DEFINITIONS <defs>. These are preprocessor definitions **contained in the interface (header files) of the component** that are set or unset anytime when this component will be used.

- for a required PID component:
  - DEPEND <dep_component>. This is the specification of the dependency. In this case the dependency is a required PID component and <dep_component> is simply the name of this component.
  - {optional} PACKAGE <dep_package>. If the PACKAGE argument is used, it means that the required component belongs to another package. In this case, <dep_package> is the name of the required package, which must have been declared as a package dependency before (in the root CMakeList.txt file of the package). If PACKAGE argument is not used, it means that the required component is part of the current package.

- for a required external component, the dependency can target either a system dependency or a custom external dependency:
  - {optional} EXTERNAL <ext_package> INCLUDE_DIRS <dirs>. When EXTERNAL argument is used, it means that the dependency is over a custom external package. In such a case <ext_package> is the name of the external package which must have been declared as a package dependency (in the root CMakeList.txt file of the package). The argument INCLUDE_DIRS is then used to specify where to find headers of the libraries used, <dirs> being the list of path to these includes, relative to the package root dir.
  - {optional} LINKS. This is an option to specify target libraries used. If used, then there must be STATIC and/or SHARED arguments used.
  - STATIC <links>. These are the static libraries used. For system libraries, system referencing must be used (e.g. -lm for libm.a).

For custom external packages complete path to libraries, relative to required external package root dir must be used.

– SHARED <links>. These are the shared libraries used. For system libraries, system referencing must be used (e.g. -lm for libm.so). For custom external packages, complete path to libraries, relative to required external package root dir must be used.

Constraints: Depending of the <type> of the component this function must be called in the adequate CMakeList.txt file (either in the **src**, **apps** or **test** folders. It must be called after the call to declare_PID_Component applied to the same declared component
Effect: this function is used to define a dependency between a component in the current package and another component (headers, static or shared library), either an external component or a PID component. This will configure the build process for the component binary (if any).

8 Examples

8.1 Package the-testpack-b

8.1.1 Root CMakeList.txt

```
PROJECT(the-testpack-b)
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.11)
set(WORKSPACE_DIR ${CMAKE_SOURCE_DIR}/../..
CACHE PATH "root of the packages workspace directory")
list(APPEND CMAKE_MODULE_PATH ${WORKSPACE_DIR}/share/cmake/system)
include(Package_Definition)

declare_PID_Package(
   AUTHOR       Robin Passama
   INSTITUTION  LIRMM
   YEAR         2013
   LICENSE      CeCILL
   ADDRESS      git@idh.lirmm.fr:perso/passama/the-testpack-b.git
   DESCRIPTION  test package B for PID
)

set_PID_Package_Version(1 1)

# adding some binary packages references
add_PID_Package_Reference(
     BINARY VERSION 0 1 0 SYSTEM linux URL
http://lirmm.lirmm.fr/FileX/get?auto=1&k=rfYTf1gkpI5XtEpQWVA
http://lirmm.lirmm.fr/FileX/get?auto=1&k=oMyg4JVeeKYWpqwEFxE)

add_PID_Package_Reference(
     BINARY VERSION 1 1 SYSTEM linux URL
http://lirmm.lirmm.fr/FileX/get?auto=1&k=DYt2j35Kw8ozOgHfoVA
http://lirmm.lirmm.fr/FileX/get?auto=1&k=zEx02N4KWfzDWPTxiO)


# from here we manage packages dependencies #
find_package (Boost REQUIRED) #system library
find_package (the-testpack-a 1.0 REQUIRED lib-b-sh lib-x)

if(NOT Boost_FOUND)
```

```
   message("FATAL_ERROR Boost has not been found
            -> impossible to build ${PROJECT_NAME} package")
return()
endif()

if(Boost_DIR-NOTFOUND)
set(BOOST_ROOT /usr)
endif()

declare_PID_Package_Dependency(
    PACKAGE Boost
    EXTERNAL ${BOOST_ROOT}
    VERSION "${Boost_MAJOR_VERSION}.${Boost_MINOR_VERSION}.
${Boost_SUBMINOR_VERSION}")

declare_PID_Package_Dependency (
   PACKAGE the-testpack-a
   PID VERSION 1 0
   COMPONENTS lib-b-sh lib-x)

build_PID_Package()
```

The package the-testpack-b (version 1.1) requires the Boost external package and the package the-testpack-a (version 1.0). It defines URL where to find binary package version relocatable archive using the `add_PID_Package_Reference` function.

8.1.2 Libraries

Libraries source code is contained in **include** and **src** sub-folders of the package. The CMakeList.txt file of the **src** folder declare available libraries:

```
#libY -> dependency with libX
declare_PID_Component(HEADER_LIB NAME lib-y DIRECTORY lib_y)
declare_PID_Component_Dependency (COMPONENT lib-y EXPORT
DEPEND lib-x PACKAGE the-testpack-a
EXPORTED_DEFINITIONS USE_EXTERNAL_LIB_X)

#libYBis -> dependency with libX
declare_PID_Component(HEADER_LIB NAME lib-y-bis DIRECTORY lib_y
```

```
EXPORTED_DEFINITIONS NUSE_EXTERNAL_LIB_X)

#libC shared -> dependency with libY
declare_PID_Component(SHARED_LIB NAME lib-c-sh DIRECTORY lib_c
INTERNAL DEFINITIONS A_VERY_SPECIFIC_IMPLEM)
declare_PID_Component_Dependency(COMPONENT lib-c-sh EXTERNAL Boost
INCLUDE_DIRS <Boost>/include)
declare_PID_Component_Dependency(COMPONENT lib-c-sh EXPORT
DEPEND lib-y)

#libC static -> dependency with libY
declare_PID_Component(STATIC_LIB NAME lib-c-st DIRECTORY lib_c
INTERNAL  DEFINITIONS NA_VERY_SPECIFIC_IMPLEM)
declare_PID_Component_Dependency(COMPONENT lib-c-st EXTERNAL Boost
INCLUDE_DIRS <Boost>/include)
declare_PID_Component_Dependency(COMPONENT lib-c-st EXPORT
DEPEND lib-y-bis)
```

The declared libraries are:

- *lib-y* a header library, whose header files are in the **lib_y** sub-folder of the **include** folder.
- *lib-y-bis* a header library, whose header files are the same as those of *lib-y* (same folder used), but whose definitions used are different (thus modifying the behaviour of the library).
- *lib-c-sh* a shared library, whose header files are in the **lib_c** sub-folder of the **include** folder, and whose source files are in the **lib_c** sub-folder of the **src** folder.
- *lib-c-st* a static library based on the same source code that the *lib-c-sh* library (same folders) but with different definitions and dependencies and built a different way (as an archive instead of a dynamic object).

shared code between lib-y and lib-y-bis libraries The folder `<package root>/include/lib_y` contains a header file named *libY.h* whose content is:

```
#ifndef LIBY_INCLUDE
#define LIBY_INCLUDE
#include <stdio.h>
```

```
#ifdef USE_EXTERNAL_LIB_X
#include <libX.h>
typedef struct{
    LibXType onetwo;
    unsigned char three; char four;
    unsigned long five;
} LibYType;
#else
typedef struct{
    char tutu [256];
    unsigned char tata[12];
    unsigned char three;
    char four; unsigned long five;
} LibYType;
#endif

typedef LibYType YBuffer[10];
void liby_print(YBuffer yb){
    for (unsigned int i = 0; i < 10;i++){
#ifdef USE_EXTERNAL_LIB_X
    #ifdef USE_LIBX_PRINT
        libx_print(yb[i].onetwo);
    #else
        #ifdef USE_LIBA_FOR_COMPUTING
        libx_print_with_liba(&yb[i].onetwo);
        #else
           #error "NO IMPLEMENTATION AVAILABLE FOR LIBX !!!"
        #endif
     #endif
     printf("three : %d, four = %c, five : %lu\n",
     yb[i].three, yb[i].four, yb[i].five);
#else
    printf("tutu : %s, tata : %s, three : %d, four = %c,
    five : %lu\n", yb[i].tutu, yb[i].tata, yb[i].three,
    yb[i].four, yb[i].five);
#endif
    }
}
#endif
```

As reader can notice, depending on the value of the USE_EXTERNAL_LIB_X preprocessor variable, this code will have different behaviours. It will notably condition the use of another library *lib-x* supposed to be defined in another package, because it condition the #include<libX.h> directive. In CMakeList.txt, the *lib-y* header library component defines this preprocessor variable while the *lib-y-bis* header library component undefines it:

- when USE_EXTERNAL_LIB_X is defined it forces the use of a dependency to another component *lib-x* of another package *the-testpack-a*. When this dependency is declared the EXPORTED_DEFINITION keyword is used to say that the preprocessor variable is defined in the interface (a header file) of *lib-y*. In this case the *lib-x* component is exported (use of EXPORT keyword in dependency declaration) since it is contained in the interface (a header file) of the *lib-y* component.

- when USE_EXTERNAL_LIB_X is not defined (using NUSE_EXTERNAL_LIB_X to negate it) the dependency to lib-x component is not defined (no #include<libX.h> directive). Consequently the EXPORTED_DEFINITION keyword is used directly into the declare_PID_Component function (see declaration of *lib-y-bis*). Indeed, this preprocessor definition cannot be attached to a particular dependency (no dependency declared for *lib-y-bis*).

This way we can associate different component to the same source code by playing on preprocessor variables. since these definition are exported they can be used by libraries that are using either version of this code (*lib-y* or *lib-y-bis*). For instance, we suppose that the *lib-x* library also exports some preprocessor variable definitions, the same way as *lib-y*: USE_LIBX_PRINT and USE_LIBA_FOR_COMPUTING. The behaviour of *lib-y* is conditional whether the preprocessor variable of *lib-x* that has been defined by the component.

shared code between lib-c-sh and lib-c-st libraries The <package root>/include/lib_c folder contains a header file named *libC.h* whose content is:

```
#ifndef LIBC_INCLUDE
#define LIBC_INCLUDE

#define LIB_C_CONST_VALUE 10.0256
extern "C"{
#include <libY.h>
}
```

```
void libc_function1(YBuffer);
void libc_function2(YBuffer);

#endif
```

The folder <package root>/src/lib_c contains a source file named *code1.cc* whose content is:

```
#include <libC.h>
#include <stdio.h>
#include <boost/math/special_functions/factorials.hpp>

void libc_function1(YBuffer data){
    liby_print(data);
    double res = boost::math::factorial<double>(data[0].five);
    printf("factorial = %lf, with factor = %lf\n", res,
            res*LIB_C_CONST_VALUE);
}

void libc_function2(YBuffer data){
#ifdef A_VERY_SPECIFIC_IMPLEM
    printf("SPECIFIC libc_function2 !!!\n");
    liby_print(data);
#else
    printf("STANDARD libc_function2 !!!\n");
    liby_print(data);
#endif
}
```

As readers can notice the interface (header files) is the same for both libraries *lib-c-sh* and *lib-c-st*: there is no use of a preprocessor variable in *libC.h* to differentiate alternative signatures. Apart from the nature of their resulting binary (archive of objects or dynamic object) their main difference is an alternative implementation of the function `libc_function2`, using preprocessor variable `A_VERY_SPECIFIC_IMPLEM` (in *code1.cc*). This variable is defined by *lib-c-sh* and undefined by *lib-c-st* with the `INTERNAL` keyword (see CMakeList.txt) since it applies only to the implementation of both component, not their interface (otherwise it would be defined with the `EXPORTED_DEFINITION` keyword, like for *lib-y* and *lib-y-bis*).

Both libraries have a dependency with the external Boost package (they need to know where are the include folders of that package) and they each define an internal dependency:

- *lib-c-sh* depends on *lib-y*, consequently `USE_EXTERNAL_LIB_X` will be defined (by transitivity).

- *lib-c-st* depends on *lib-y-bis*, consequently `USE_EXTERNAL_LIB_X` will be undefined (by transitivity).

Finally both libraries will have slightly different behaviours depending on either (1) their local implementation and (2) their dependencies.

Finally readers can also notice that dependencies are exported or not:

- Dependency to external Boost package is not exported since the `#include <boost/math/special_functions/factorials.hpp>` directive is only in the *code1.cc* source file.

- Dependencies to *lib-y* or *lib-y-bis* are exported since the `#include <libY.h>` directive is in the interface (*libC.h*) of both libraries.

8.1.3 Applications

Applications source code is contained in the **apps** sub-folder of the package. The CMakeList.txt file of the **apps** folder declare available applications:

```
#1 first version (example) of application
declare_PID_Component(EXAMPLE_APPLICATION NAME app-b1
     DIRECTORY app_B1)
declare_PID_Component_Dependency(COMPONENT app-b1
     DEPEND lib-b-sh PACKAGE the-testpack-a)

#2 second version of application
declare_PID_Component(APPLICATION NAME app-b1-evolved
     DIRECTORY app_B1)
declare_PID_Component_Dependency (COMPONENT app-b1-evolved
     DEPEND lib-b-sh PACKAGE the-testpack-a)
declare_PID_Component_Dependency (COMPONENT app-b1-evolved
DEPEND lib-c-sh  INTERNAL_DEFINITIONS EVOLVED_IMPLEM)
```

The declared applications are:

- *app-b1* an example application, whose code is placed into the `app_b1` sub-folder of the **apps** folder.

- *app-b1-evolved* a standard application which share its source code with *app-b1* (its source code is also in `app_b1`).

The folder `<package root>/apps/app_b1` contains a source file named *app_ b1.cc* whose content is:

```
#include <stdlib.h>
extern "C"{
#include <libB.h>
}

#ifdef EVOLVED_IMPLEM
#include <libC.h>
#endif

int main(int argc, char* argv[]){
#ifdef EVOLVED_IMPLEM
libb_function1(libb_function2());

YBuffer buff;
for (unsigned int i = 0;i < 10 ; i++){
buff[i].three = i+1;
buff[i].four = 'c';
buff[i].five = (unsigned long) buff[i].three * 2;
}
libc_function1(buff);
libc_function2(buff);
#else
libb_function1(libb_function2());
#endif
return 0;
}
```

As readers can notice source code contains alternative implementations the same way as for *lib-c-sh* and *lib-s-st* libraries:

- *app-b1* is the simple version, `EVOLVED_IMPLEM` is let undefined (equivalent as going `INTERNAL_DEFINITIONS NEVOLVED_IMPLEM`). It is just a simple example of usage of library lib-b-sh.

- *app-b1-evolved* is a more complex version. Readers have to notice that application **having no interface they cannot export any definition**. Contrary to libraries, applications declarations can so only use `INTERNAL_DEFINITIONS` and `IMPORTED_DEFINITIONS` keywords.

Both applications define a dependency to *lib-b-sh* library of the package *the-testpack-a*. *app-b1-evolved* defines an additional dependency to *lib-c-sh*. The usage of this library is conditional (in source code) depending on the `EVOLVED_IMPLEM` preprocessor variable (that guards the `#include<libC.h>` directive). For *app-b1-evolved*, call to `declare_PID_Component_Dependency` so uses the `INTERNAL_DEFINITIONS` keyword to define `EVOLVED_IMPLEM`.

8.2 Package the-testpack-c

POUBELLE

CATEGORIES

The workspace also provides categories : each category defines a specific subject that helps classifying packages/frameworks. For instance the category sensor will reference all packages or frameworks that implement the driver of a sensor. Categories can be hierarchically refined into sub categories for instance the category camera refines the category sensors. Categories can be viewed as lists of links that target frameworks/packages. The process of classification of packages is itself coming from package description (i.e. a package specifies the categories it belongs to).

the **categories** folder is structured according a set of topics, each topic being itself a folder that in turn contains either other subcategories or **links to frameworks**. **categories** only goal is to help finding available packages (under binary form) according to a given theme. Considering a category, packages that are relevant for the theme it defines are referenced as links (symbolic relative links in Linux) to frameworks folders.

When developing it is often useful to know which package provide some useful things relative to one or more topics. The aim of the **categories** folder is to standardize the classification and the finding of packages relevant for given concerns.

The **categories** folder is structured into subfolders that describe known categories. Each category folder is itself structured the following way:

- sub folder describes sub categories. A sub category refines the topic of its containing category. For instance folders **arms** and **wheeled vehicles** are contained in the category folder **robot**. These sub categories can in turn be refined the same way.

- symbolic links represent target packages of the category. The name of the link is simply the name of the package, the target of the link is a given package framework. The target itself is always expressed as a **relative path** from the containing category folder to the target framework. By default these links target nothing in the file system as long as the package's framework has not been installed in the **frameworks** folder.

Remark: a same package can be targeted by more than one link: it means that the package belongs to different (sub)categories according to the different point of view of the developers. Doing so, anyone can have his own point of view of the package depending on its center of interest.

To organize the development, defining some "standard categories" should help developers to classify their packages. This requires to list categories of code that is useful for robotic applications development. For instance :

- sensors
  - vision
    - cameras
    - kinects
    - lasers
  - force sensors
  - encoders
- robots
  - arms
  - wheeled vehicles
  - humanoids
- simulators
- navigation
  - cartography
  - positioning
- planning
  - mission
  - path
  - trajectory
- middleware
  - ros
  - contract
  - orocos

Of course categories can evolve/change along time, when developers want new categories.

Working with categories

The only people that can modify **categories** folder are package administrators. Each time a new package is created, developers define which are its relevant categories and wrote them in the package.manifest file of the package. When an administrator of a package wants to reference it :

- he first copy/rename the package.manifest file into the **references** folder.
- according to the relevant categories of tha package defined in the manifest, he:
  - creates new (sub)category folders in the the **categories** folder of its workspace, if the corresponding categories do not exist.
  - creates as many symbolic links that target the package framework as there are relevant categories for the package and put these links into the adequate category folders.
- once done he commits its workspace.

From developers point of view, categories just help finding packages that are relevant for given topics. Each time they update their workspace, all available categories and classification of packages according to categories are updated.

To search packages, they can simply navigate in the file system of the **categories** folder.

- if the package is already installed in the **frameworks** folder he can simply follow the link.
- otherwise the link is broken, and the developer has to install the corresponding package.